
**Universidad Carlos III de Madrid**

Escuela Politécnica Superior

PhD Thesis

# Application partitioning and mapping techniques for heterogeneous parallel platforms

Leganés June 2016

**Author:** Rafael Sotomayor Fernández

**Advisor:** J. Daniel García Sánchez

PhD Thesis

# Application partitioning and mapping techniques for heterogeneous parallel platforms

**Author:** Rafael Sotomayor Fernández

**Advisor:** J. Daniel García Sánchez

Presidente D. .....

Vocal D. .....

Secretario D. .....

Realizado el acto de defensa y la lectura de tesis en ..... el día ..... de ..... del ao 2016.

Calificación: .....

El Presidente El Secretario

Los Vocales

# Abstract

In recent years, performance gains provided by clock and ILP techniques have considerably slowed down. As a result, parallel programming has become the dominant programming paradigm used to improve performance in multi-core devices. In line with this, parallel use of specialized accelerators has started gaining importance. However, adapting legacy source code in order to make use of these technologies is a time consuming and error prone task, requiring specialised knowledge.

The main goal of this Thesis is to simplify the task of transforming sequential legacy code into parallel code. This code will be capable of making full use of the different computing devices that an heterogeneous parallel platform can have, such as modern CPUs, GPUs, FPGAs, and DSPs. With this, it is possible to improve sequential code based on different criteria, such as time performance.

As a result, we propose an architecture description language to describe heterogeneous parallel platforms. We suggest a new software annotation syntax to describe the behaviour of the code from a high-level point of view while preserving its maintainability, along with automatic annotation techniques. Finally, we propose a set of task partitioning techniques to split the code and execute it in parallel using the available computing devices. Results aim to demonstrate that the proposed techniques can be applied to different accelerator devices and source code, and that the chosen metrics are improved with respect to the original sequential code.

# Resumen

En los últimos años, el rendimiento derivado de la frecuencia de reloj y de las técnicas de ILP se han reducido considerablemente. Como resultado, la programación paralela se ha convertido en el paradigma predominante a la hora de mejorar el rendimiento en dispositivos multi-core. A raíz de esto, el uso en paralelo de aceleradores especializados a empezado a cobrar importancia. No obstante, adaptar código legado para hacer uso de estas tecnologías es una tarea tediosa y proclive a errores, para la que se requiere un conocimiento muy específico.

El objetivo principal de esta Tesis es simplificar la tarea de transformar código legado secuencial en código paralelo. Este código podrá utilizar los distintos dispositivos de cómputo disponibles en una plataforma paralela heterogénea, tales como CPUs, GPUs, FPGAs o DSPs. Así, es posible mejorar código secuencial en base a distintos criterios, como el rendimiento.

Como resultado, se propone un lenguaje de descripción de arquitecturas para describir plataformas paralelas heterogéneas. Se sugiere una nueva sintaxis de anotación del software para describir el comportamiento del programa desde un punto de vista de alto nivel, preservando su mantenibilidad, así como técnicas de anotación automática. Finalmente, se propone un conjunto de técnicas de particionamiento para dividir el programa y ejecutarlo en paralelo haciendo uso de los distintos dispositivos de cómputo. Los resultados buscan demostrar que las técnicas propuestas se pueden aplicar a distintos aceleradores y códigos fuente, y que las métricas escogidas mejoran con respecto a las del código secuencial original.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In recent years, traditional approaches for boosting CPU, such as increasing clock speed and execution optimization, have started giving diminishing returns. This is because of physical constraints, such as heat dissipation, energy consumption, and current leakage.

Regardless of this, Moore's Law continues to hold, and the gains in transistor density have remained over the years. This has lead to the creation of parallel architectures of greater processing capabilities, such as multi- and many-core processors, where many CPUs sit on the same die. Several techniques and features that complement this model have appeared, such as hyper-threading or vector instructions.

Furthermore, the tendency has shifted from a single processing element, to heterogeneous architectures that combine different kinds of computational elements, such as CPUs, GPUs, FPGAs, and DSPs. They are all specialized for certain tasks and typically have their own programming models.

This chapter introduces the groundwork of this Thesis. Section 1.1 explains the motivation for this work in the frame of the aforementioned context. Section 1.2 contains a definition of the goals, requirements, and expected contributions to be achieved during the development of the Thesis. Lastly, the structure of the rest of the document is outlined.

## 1.1 Motivation

The aforementioned computational elements can provide a massive improvement in performance, at a small cost in terms of power consumption [82]. However, writing efficient parallel source code that copes with those hardware platforms has a significant cost in terms of the learning curve and required development time.

Typically, developers have to identify potential SIMD problems in their code that are fit for parallelization, and then rewrite them in an architecture-specific language, such as OpenACC and CUDA. In order to achieve good performance, a developer has to make significant changes in order to exploit the accelerator capabilities of the

processor. Such programming complexity is a barrier to greater adoption of heterogeneous parallel platforms. First, each family of devices has a different hardware and software architecture, and it is usually necessary to implement applications using a specific programming model. This makes it very difficult to write code that makes full use of these heterogeneous architectures. Second, a very intimate knowledge of both architectures and programming models is necessary to make an efficient use of these devices with regards to high performance. The purpose of this Thesis is to develop a unified framework that can be used in this kind of heterogeneous parallel platforms in order to: (1) reduce power consumption, (2) improve performance, and (3) increase productivity realizing designs.

A common solution for targeting the application development in heterogeneous platforms is the usage of compilers and tools [120], which automatically parallelize the code. However, there are still no wide-spread solutions that perfectly transform sequential source code into optimized parallel code [93]. Automatic tools can accelerate the source-to-source transformation in many cases [77]. Nevertheless, both performance and portability could be targets of this transformation process [118].

A desirable feature of the auto-parallelization tools is the detection of parallel code regions, called kernels. If kernels are identified, it is possible to take advantage of current high-level parallel programming abstractions such as parallel algorithms, skeletons, and parallel patterns, and not low-level solutions such as CUDA or OpenCL.

## 1.2 Goals

In light of the growing necessity for parallel programs, as well as the complexity of modern parallel programming models, we define the following goal for this Thesis:

> **Goal**: The main goal of this Thesis is to optimize code execution based on time performance. To this end, sequential code will be partitioned into the processors of an heterogeneous parallel platform, so that it can be run in a parallel fashion. The platform may be composed of one CPU and any combination of other parallel devices, such as GPUs and other accelerators.

This goal can be divided into the following specific goals:

1. **To define a hardware model** that represents heterogeneous parallel platforms with sufficient level of detail. This model will be used to check the software for devices, capabilities, communication systems, etc.

2. **To define an programming model** that will be the base for the software partitioning. It will be necessary to describe the interaction between the different devices in order to partition the software and orchestrate the data movements.

3. **To create an specification for annotating software.** This specification will allow to annotate the code in order to give semantic information regarding the partitioning. It will also be used to optimize the parallel code whenever possible.

4. **To develop automatic annotation techniques.** These techniques will reduce errors and fails when annotating and later transforming the code.

5. **To develop a set of static task partitioning techniques.** These techniques will be based on the execution model, and will be used as the basic case for the full partitioning algorithms. They will be used to partition the code statically based on the static information.

These sub-goals will fulfil the following requirements that are needed to achieve the main goal:

1. **A description of the underlying hardware** and its capabilities with enough level of detail. Given that the code will be optimized based on the platform where it will be run, it is necessary to know the details so as to be able to adapt.

2. **A description of the characteristics of the software.** Because each kind of processor is better suited to run certain software, it is necessary to find the characteristics of the code in order to find the best match for it.

3. **A set of task partitioning techniques** that will take all the information mentioned above and will assign different pieces of software to different processors in order to optimize time performance.

## 1.3 Overview

The task partitioning techniques are based around executing pieces of parallel in the different devices available in a heterogeneous platform.

Each device has strong points that make them better for solving specific problems. The performance obtained depends on certain characteristics of the code, as well as the physical constraints of the device. Therefore, it is necessary to know in detail the underlying hardware and the prospective parallel code. We identify three necessary steps, as shown in Figure 1.1:

1. **Detection of hardware capabilities.** It is important to know what devices are available, as well as their capabilities (e.g. memory size is important when offloading a piece of code to a specific device). Therefore, it is necessary to dynamically detect the underlying hardware capabilities.

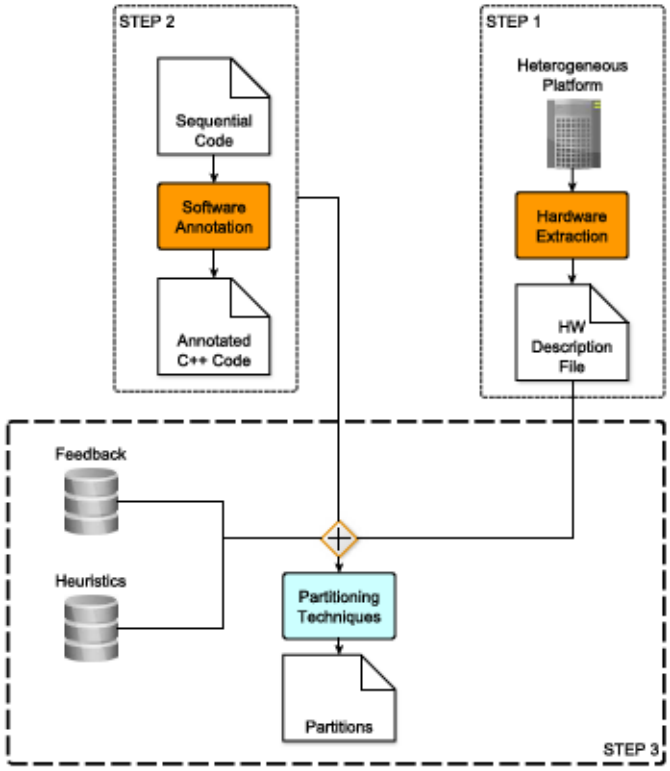


Figure 1.1: Proposed steps for partitioning software.

2. **Detection of software characteristics**. This characterization is loosely based on high-level parallel patterns, such as pipelines, farms and maps [88, 86]. It allows the efficient orchestration of the code, as well as several optimizations.

3. **A scheduling policy** that orchestrates the software execution on the detected hardware. Depending on the hardware capabilities and the code orchestration, this policy will obtain the best configuration in terms of performance.

These steps depend on the underlying execution and memory models that have been devised. For example, the memory model affects how the memory hierarchy is defined in the hardware detection process. Also, the code orchestration that applies to both the software characterization and the scheduling algorithms depends directly on the execution model.

## 1.4 Document structure

The rest of this document is structured as follows: Chapter 2 shows the state of art in the different technologies and techniques that have been investigated and

developed in this Thesis. Chapter 3 explains the proposed Hardware Description language, as well as the underlying models where the rest of the work for this Thesis is based. Chapter 4 details the mechanisms to identify parallel code, as well as the the definition of the annotation semantics. In Chapter 5 we discuss the software partitioning and scheduling techniques applied in this Thesis. Chapter 6 provides the evaluation process of the work. Lastly, Chapter 7 lists the conclusions drawn from the results, as well as different options to expand the work so far.

# Chapter 2

# State of the art

There are some key points to this Thesis that serve as a base for the rest of the work.

One of the basis of this Thesis will be the usage of heterogeneous parallel platforms. These platforms will contain processors of different characteristics. At least, a CPU will be present, and there can be any combination of computational accelerators, such as GPUs. Section 2.1 summarizes a list of the devices considered for the Thesis, as well as some of the most recent hardware developments in parallelism.

In order to be able to utilise the heterogeneous parallel platforms, it is necessary to know what components it contains. This includes devices, memory banks, and links. There are several hardware description languages that can be found in the literature. A small overview can be seen in Section 2.2.

One primitive way to introduce parallelism is to use native threads. However, modifying a sequential code in order to include threads is a time-consuming and error-prone task. The process of redesigning and transforming the code requires that users manually control certain problems inherent to parallel programming models, like race conditions. To solve these problems, many parallel frameworks have appeared during the last years. These frameworks typically aim to provide parallelism from a high-level point of view, while hiding particular details of the implementation. Section 2.3 shows a list of recent parallel frameworks of different characteristics that offer different APIs in order to provide parallelism.

As said in the previous chapter, developing and/or transforming the code in order to make an efficient use of the available parallel processors is a time-consuming and error-prone task, where users require intimate knowledge of the underlying hardware and the associated programming models. Therefore, although not a core functionality of this Thesis, automatic code transformation is considered an important asset to have as well. In Section 2.4, we will also discuss the state-of-the-art regarding code transformation for multi-device parallel frameworks.

Lastly, the core of the work in this Thesis are the code partitioning techniques. These techniques are used to optimize performance by running pieces of code in parallel using the available devices. In Section 2.5 we recapitulate some of the most

recent works about application partitioning.

## 2.1 Parallel processors

As explained, one of the main pillars of the Thesis is the use of heterogeneous parallel platforms. These platforms contain devices with different strengths and capabilities that are fit for software of certain characteristics. When used in combination, they can help to accelerate code that may otherwise be infeasible or extremely complex [124], providing performance that a single device cannot match.

There exist many different devices that can be used, called hardware accelerators. However, for the sake of feasibility, we will limit this work, and therefore this study, to three families of devices: CPUs, GPU, and FPGA.

### 2.1.1 CPU

In recent years, chip design has reached a limit in clock frequencies due to heat dissipation, power usage, and stray voltage [115]. At the same time, traditional Instruction Level Parallelism (ILP) techniques have hit a wall of their own. These techniques include instruction pipelining, out-of-order execution and register renaming, to list a few.

Because of this, efforts have been placed towards the development of processors that can execute of multiple instructions in parallel. An entirely different programming model as been developed to make use of this new architecture. We distinguish between two different trends in parallel programming: thread level parallelism and data level parallelism

**Thread Level Parallelism** (TLP) is the most extended technique used to take advantage of parallel hardware features. In TLP parallelism is achieved by concurrently executing several threads or workers.

Modern computers architectures include many features to improve TLP. Intel's Hyper-Threading technology [64] is one example specific to Intel's Core processor family. The idea is to include two logical processors in a processing unit, where each keeps their own register state, but share some of the execution resources such as the caches and buses. Other example is the ccNUMA (Cache Coherent Non-Uniform Memory Access) architecture. In this architecture, multiple cores have their own memory banks, in order to benefit from locality access, but all cores share a unified vision of the memory.

All these mechanisms help to increase the performance if the application is split into several tasks. Every task is typically executed by a single thread, and all the threads are executed in parallel. The main challenge for the programmer is how to efficiently split the application into tasks, in such a way that they can obtain a high degree of parallelism. The tasks must be equally weighted in terms of computational cost to obtain the maximum performance. Moreover, the programmer has to deal

with the issues derived from managing data shared between tasks in a coordinated way. Usually, this involves a memory access policy that allows exclusive accesses for a task.

**Data Level parallelism** exploits parallelism by executing the same operation over each one of the elements of a big set of data in parallel. This is known as the Single Instruction Multiple Data (SIMD) approach. Modern computers include many features to improve DLP. One of the main approaches towards improving DLP is actually the use of specified accelerators, such as the GPUs, which are described in the following subsection. Another option is to enhance the CPU with data parallel units that are used via special vector instructions. These instructions, such as the SSE family or the AVX family, permit to execute the same operation over several values simultaneously.

### 2.1.2 Graphic processing units

Graphic processing units (GPUs) are specialized devices that use a fixed pipeline in order to handle the graphic demands of applications in certain areas, such as video games, CAD and 3D applications. They are specially well suited to solve the so-called pleasingly parallel problems, where there is no dependency between the different parallel tasks. The aforementioned SIMD problems, which can be solved by performing the same operation over a bit set of data, is a classic example of pleasingly parallel problems, and the main niche for GPUs.

With the advent in 2001 of floating point support and other features on the devices, GPUs become a common choice, not only for applications in graphic areas, but also in more general applications that were typically handled by the CPU. The use of the GPUs for general purposes gave birth to the term **General Purpose Computing on Graphics Processing Units** (GPGPU).

These devices have been used in multitude of areas, from evolutionary algorithms [83, 84] to scientific computing like DNA sequencing [121], light propagation [7], and weather prediction [89].

In order to use GPUs for general purposes, special programming models are required. The two most extended ones are CUDA [97] and OpenCL [98]. Although the focus of this Thesis on heterogeneous platforms, CUDA is not of interest for this study. The reason is that CUDA is specific to NVidia graphic cards. OpenCL is explained in detail in Section 2.3.

One major disadvantage of the GPUs and, indeed, of all external accelerators, is the data transfer overhead. Although some GPUs are integrated inside the motherboard or the same die as the CPU and share the memory space, it is typical to have an external GPU device connected through a PCI or PCI Express bus. When users have to move data from the CPU to the GPU and back, they incur in a transfer overhead due to the limitations of the bus. This is an important factor to consider when choosing between the CPU and the GPU.

Another potential problem of the GPUs is the branching. Because of the way GPUs group the threads, and the way the threads operate, it is possible that branching lowers the performance considerably, although this is not always the case, and the compiler usually optimizes the code so that this does not happen.

### 2.1.3 Field-programmable gate array

A field-programmable gate array (FPGA) is an integrated circuit based around a matrix of configurable, interconnected logic blocks. One of the main advantages of FPGAs is that parts of their circuitry can be reconfigured dynamically by the user as needed, hence the name field-programmable.

FPGAs are different from the application-specific integrated circuits (ASIC), which are designed for a specific use. An example of an ASIC would be an SSL transactions accelerator. Because ASIC are custom-manufactured, FPGAs are usually more affordable, facilitating, for example, large-scale integration of FPGAs [73].

FPGAs use programmable logic elements, called look-up tables (LUTs), in order to implement simple logic functions. These tables can perform any logical function of a fixed amount of inputs or outputs. In order to extend this limitation, several tables are used. The tables are connected via an interconnection matrix, where each table is surrounded by programmable interconnections [87].

As explained, one of the advantages of FPGAs is their reconfigurability. In particular, modern circuits have reconfigurable LUTs, reconfigurable I/O blocks, and reconfigurable, dedicated memory blocks. Modern FPGAs allow static and dynamic partial reconfiguration, where the dynamic version is applied on a particular part of the device while the rest of the circuit is still running [127]. The circuit may have other fixed elements, known as hard blocks, but those are used to increase speed in exchange for the reconfigurability.

In order to use an FPGA, one needs to define its behaviour with an architectural description language (see Section 2.2 for more on hardware description languages). Common languages for this are VHDL [92] and Verilog [12]. Both have a detailed syntax that is used to define the behaviour of different elements of the circuit, such as signals, I/O ports and so on. Although efforts have been made recently for FPGAs to provide support to high-level programming languages, many people still find it easier to use a different accelerator, or even a CPU [105].

Despite the difficulty inherent to the programming model of FPGAs, their architecture ensures that they can usually solve computable problems faster than other devices, because their components can be optimized ad-hoc through reconfiguration. Their used is extended, from aerospace applications [75], to consumer electronics [128].

## 2.2 Architecture description languages

The techniques developed for this Thesis must be generic, and, for this, they must work on any heterogeneous hardware architecture. To this end, it is necessary to identify the hardware elements and their characteristics, within the limits of the devices mentioned in the previous Section.

There exist many parameterized architecture description languages (ADL) that perform this particular task, typically used for embedded systems design. They can be classified according to the target of the language [91], ranging from purely structural, which describe the general hardware architecture, to behavioural, which also contain information about instruction semantics, as well as combinations in-between.

A simple example is Sequoia [47]. It is a programming model for multi-core focused on explicitly representing the memory hierarchy as a tree based on the Parallel Memory Hierarchy [8]. It is interesting because it is a detailed description language. Its disadvantage is that it is focused on a few families of processors, thereby limiting its extensibility.

Another option is PEPPHER [20, 43], a C/C++ model that aims to facilitate the creation of performance-portable applications. PEPPHER makes use of a platform description proposed by Sandrieser et al. [108]. This description is interesting, because it is actually used to model GPUs.

These two examples are tools that make use of ADLs, but are not designed specifically for that. Additionally, the Portable Hardware Locality (hwloc) tool [28, 116] is specifically designed to generate a portable abstraction of the whole architecture, from processors and memory hierarchy to connections such as PCI buses. It also detects certain accelerators, which is an enticing bonus for its use on an heterogeneous parallel platform.

Two relevant ideas can be drawn from what has been shown so far. The first is the tendency to represent the components of an architecture as a hierarchy of elements, where each element "is contained" by the one above it. In this way, the platform, as the root node of the hierarchy, contains a set of components, such as the CPU or a GPU, as well as connections between said components, as can be USB, SATA or PCI buses. The second idea is the definition of "capabilities" as the most basic pieces of information. This affects how the model is defined. For example, a local L1 cache of size X is not a capability of a core, but rather the cache memory is a component contained in the core, and has certain capabilities, such as a size of X.

It is important to note that, as a rule of the thumb, the wider the coverage of the tool in terms of variety, the less detailed the information provided will be. For example, **hwloc** represents locality, but does not give much information about the characteristics of each particular node. In contrast, the **lshw** tool [46] is a clear example of what we are looking for. Though it is restricted to CPUs, the amount of

information provided opens the possibility to optimize the code accordingly further down the line. This is specially relevant for devices such as FPGAs or DSPs.

## 2.3 Parallel frameworks

One of the purposes of this Thesis is to provide a simple mechanism to introduce parallelism in legacy C++ software, while preserving the maintainability of the code. To this end, we consider different parallel frameworks based on whether or not they allow to preserve the original source code. We divide them in the following categories: library-based programming frameworks, annotation-based programming frameworks and the Cilk family of frameworks. A summary of the characteristics of the listed frameworks can be found in Table 2.1.

### 2.3.1 Library-based programming frameworks

#### FastFlow

FastFlow was created in 2009 by researchers of the Universities of Pisa and Torino [4]. Over the last few years, new features have been added to the framework, such as high-level patterns like the parallel for [40] or support for multiple devices [6, 41].

FastFlow is a general-purpose parallel programming framework [122]. It supports streaming and data parallelism through the use of parallel algorithmic skeletons. They rely on lock-free communication queues [3] that allow for fine grained parallelism. It works on different devices, such as GPUs, FPGAs or DSPs.

The FastFlow stacked and layered architecture 2.1 provides different levels of abstraction via C++ templates that an application has access to. They are designed as follows:

1. The **building blocks**. The lowermost level. This layer contains the basics to build the core patterns and obscure the low-level details of the concurrency mechanisms. It includes containers for the threads as well as communication mechanisms, such as the aforementioned queues. This layer is also used to obscure access to the different processors, including the CPU and other accelerators.

2. The **core patterns**. This layer is built on the building blocks. It contains the most basic, data-centric parallel patterns. These are the farm of tasks and the pipeline, as well as a feedback mechanism to allow for cyclic process graphs. Each node of these graphs would be a process or thread that would run the tasks it is fed.

3. The **high-level patterns**. This layer is built on top of the core patterns. It provides high-level parallel patterns to the user, such as the parallel-for or

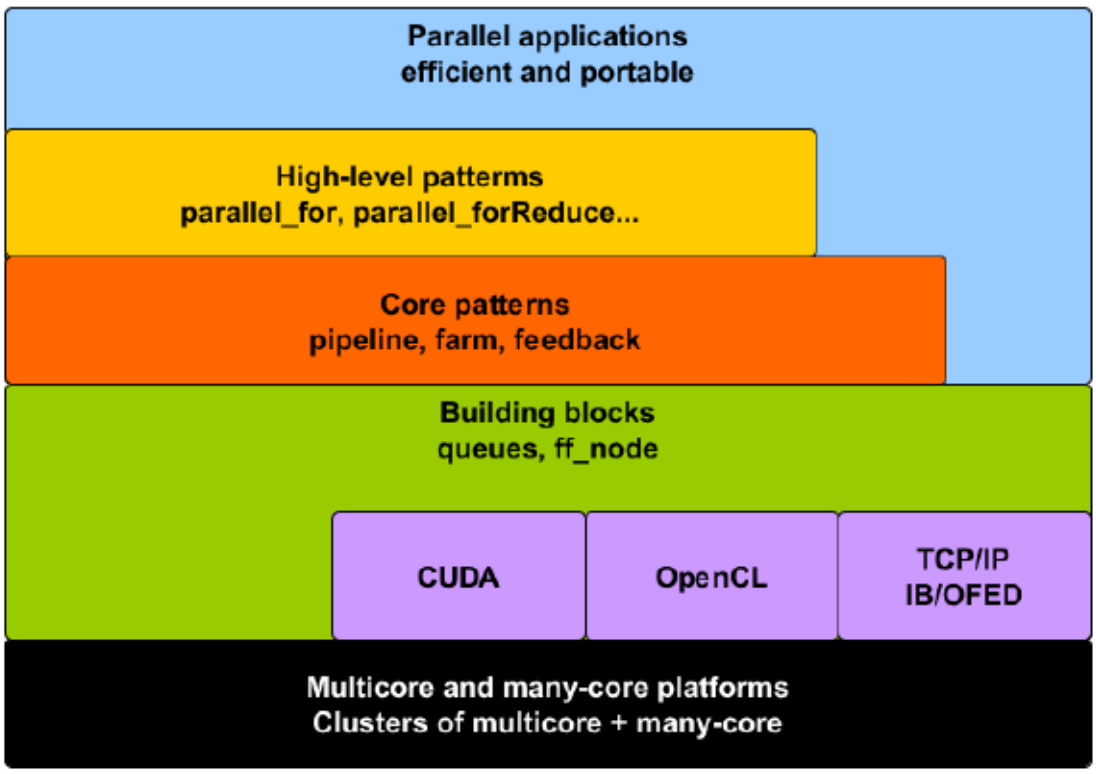


Figure 2.1: FastFlow layer stack.

Source: http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:architecture

the stencil-reduce. They can be used to parallelize sequential code, and are already optimized.

### Intel Threading Building Blocks

Intel Threading Building Blocks (TBB) is a standard C++ template library developed by Intel [69] for multi-core architectures. It provides templates for parallel algorithms, such as `parallel_for` or `parallel_reduce`, as well as parallel containers, such as `concurrent_queue` or `concurrent_vector`. It implements its own work-stealing scheduler that dequeues tasks from a FIFO queue [103].

In order to utilise TBB, one typically has to re-design the code in order to fit the available patterns. Although in these cases it often offers great performance, when the programmer is not coding an application from scratch, the code has to be modified in order to make use of the features available in TBB. Furthermore, unlike FastFlow, there is a limit to how far one can customize their code, since there is a limit to how the algorithms can be combined, and the scheduler cannot be customized.

#### OpenCL

Open Computing Language (OpenCL) [98] is a standard extension of C/C++. It allows to write SIMD kernels that can be compiled just-in-time and executed on different processors (typically CPUs and GPUs) without recompiling the code. It can also generate specific binary files for each specific architecture.

OpenCL provides a master-slave programming model, where a host device can choose to offload the execution of a parallel piece of code (called kernel) to an external accelerator. These accelerators may be any device with an OpenCL driver, such as GPUs, some FPGAs, Intel's Xeon Phi or the CPU itself. In this model, each devices, whether it be the host or an accelerator, has its own memory space, and it is possible to send and receive data from the host to an accelerator and vice-versa.

There are many similarities between the model of OpenCL and that of NVidia's CUDA [97], with the main difference being that CUDA is designed to work only with GPUs.

### 2.3.2 Language extensions

#### OpenMP

OpenMP first appeared as a C++ framework in 1998, having first appeared as a Fortran framework the year before, and quickly became a standard for most compilers. Version 3.0 saw the appearance of task parallelism, along with certain improvements to the data reduction functionality [68]. Lastly, the standard 4.0 included support for accelerator devices, although only recently have compilers started to support this standard [25].

Open Multi-Processing (OpenMP) [38] is an open specification that defines a language extension for task parallelization. The API is very simple for the user, and it is based on the use of compiler directives. OpenMP includes parallel loops, parallel regions that are executed by all the cores, and support for shared variables, among others.

OpenMP yields great performance in most scenarios, and it is very simple to use it from the sequential code because, unless one wants to do a very specific optimization, the only changes to the code are the pragmas. OpenMP offers to the user the chance to change between three thread scheduling policies. However, in part because of this, it does not naturally support more than the simple data-parallel scenarios, and it often falls to the user synchronize the threads or even modify the code in order to include a specific optimzation [31].

#### OpenACC

OpenACC first appeared as a standard with little support from compilers, although in the latest years, efforts have been made to merge it with OpenMP. For now,

major OpenACC support comes from the PGI compiler [101], although there is an experimental release of OpenACC with GCC 5.1 [51].

Open Accelerators (OpenACC) [35] is, much like OpenMP, a set of compiler directives used to orchestrate parallel regions of the code. However, unlike OpenMP, its original purpose is to offload these code regions to a variety of GPUs. To this end, the compiler transforms annotated code to low-level CUDA or OpenCL code.

The OpenACC API hides many of the low-level details about using accelerators from the user. These details include tedious work, such as data initialization, as well as more complex decisions, such as those concerning data movements between the host and the accelerators. Although the default mode of operation obtains great performance in most scenarios, the OpenACC API allows the user to override its decisions regarding code and data orchestration.

### Cilk

Cilk was first developed at MIT in 1994 as an extension to C language [24]. Over time, and with the emergence of multi-core processors, new version were released with new features to go with the flow. Cilk Arts would see the light in 2006, and Cilk++ in 2007, though both were proprietary software. Lastly, on 2009, Intel bought Cilk and made it into an Open Source project that in 2010 would be released as the current Cilk Plus [65].

Cilk refers to a family of frameworks, Cilk, Cilk++ and Cilk Plus. Although they are not annotation-based frameworks, they are also extensions of the language with very little impact on the original code and, therefore, they are consider similar in nature to the previous frameworks.

Cilk is a C language extension that allows to exploit recursive parallelism by asynchronously spawn functions based on the fork-join model. It can also specifically run parallel for loops. It is limited to multi-core architectures, but in those it shows great efficiency for multiple parallel problems. Cilk provides three simple keywords for its use:

- The `cilk_spawn` keyword is applied immediately before a function call. It indicates that the function must be run asynchronously from the caller. This
- The `cilk_sync` keyword is used as a barrier to indicate that a the function where the sync is called cannot proceed until all previously spawned functions have finished and returned to the caller.
- The `cilk_for` keyword can replace the for keyword. It allows implementing a parallel map over the iterations of the marked loop.

Of special interest is the array notation provided by Cilk [66], which allows to refer to parts of an array based on: the lower bound of the section, the length, and the stride.

An example of the array notation can be the following: `A[0:5:2]= 5;`. This annotation is equivalent to `for(i = 0; i < 10; i += 2)A[i]= 5;`

Table 2.1: Summary of the characteristics of the parallel frameworks.

| Frameworks | API | Standard | Portable | Multi device | Keeps legacy code |
|---|---|---|---|---|---|
| FastFlow | Templates | Yes | Yes | Yes | No |
| TBB | Templates | Yes | Yes | No | No |
| OpenCL | Keywords + library | No | Yes (not efficiently) | Yes | No |
| OpenMP | Pragmas | Yes | Yes | Yes | Yes |
| OpenACC | Pragmas | No | Yes | Yes | Yes |
| Cilk | Keywords | No | Yes | No | Yes |

## 2.4 Code analysis and transformation

One of the goals of this Thesis is to annotate code in order to add semantic information without having to modify or refactor the code excessively. The previous section showed information about many different kinds of frameworks. Of them, those based in annotation mechanisms fit into the aforementioned goal.

In the literature we can find many works that take advantage of open standards and frameworks in order to execute legacy code block in GPUs. Many of these works rely on compiler directives, or pragmas, in order to introduce parallelism in the code. Wienke et al. [126] use OpenACC to annotate the code and later run it on GPUs, whereas Bertolli et al. [21] combine the newest version of OpenMP 4.0 with the compilation toolchain that is Clang + LLVM.

However, those open standards are not fully supported at this moment in common open source compilers. For example, currently GCC 5 includes OpenMP 4.0 and preliminary support of OpenACC 2.0, but it is limited to specific accelerators: Intel Xeon Phi and Nvidia PTX.

One alternative to pragmas are the C++11 attributes. These attributes are described in the C++11 standard [70]. C++11 attributes are expressions placed between double brackets that can be used to add specific information about a syntactic entity These annotations will refer to the syntactic entity (e.g. statement, variable definition, loops), that usually is placed immediately after the annotation.

From a semantic point of view, C++11 attributes provide four advantages over pragma-based frameworks. First, a code with C++11 attributes does not need support from the preprocessor to include additional information. Second, C++11 attributes differ from existing syntaxis, (e.g. `__attribute__`, `__declspec`, and `#pragma`) by being applicable, essentially, to every syntactic element in the code. Third, attributes may be supported either by the compiler or by external tools, making evolution much simpler (e.g. attributes may be used as a mechanism for defining domain specific languages). Finally, in contrast with previous similar mechanisms, the usage of attributes provides a portable way for annotating source code.

The most widely spread open source C++ compilers (i.e, GNU g++ and Clang) support C++11 generalized attributes. For example, Barath et al. [16] propose two new C++11 attributes to improve move semantic optimizations. A standard compiler will ignore these attributes, but if the compiler includes them, they can be used to optimize all the copy/move operations.

There are more annotation mechanisms for heterogeneous platforms, such as the one provided with StarSs [30, 14], but expanding more on this venue does not provide more add to the discussion.

Another interesting point to consider is the concept of **automatic annotation**. Automation of the different processes developed in this Thesis is a desirable, if not necessary improvement. As a starting point, there are several research works that address the discovery of potential parallel code in sequential legacy code. Current approaches can be classified in terms of the code analysis carried out, static and dynamic, and the presence of guided parallelism, automatic, and semi-automatic. Most of the works are based on OpenCL-like models and, therefore, around the discovery of parallel kernels.

In the topic of static code analysis, most compiler-based tools are based, or make used, of the polyhedral (or polytope) model for code optimization. Although modifying the compiler to transform the code is not inside the framework of this Thesis, it is still an interesting technique to know for future reference.

The polyhedral model [57] is a technique for optimizing nested loops, which involves overhead reduction, or locality optimization. In particular, the polyhedral model represents each iteration as a node inside a polytope, in order to perform transformations such as tiling or skewing.

Most of the polyhedral tools that can be found in the literature are based on Poly [58], specially in the cases where the target architecture are multi-core processors. One example of Poly-based tools is the Clan tool [19]. Another Poly-based tool for nested loop optimization is Pluto [27]. It has been used in many solutions [26], including C-to-CUDA code generation [15, 90, 18]

Polyhedral, and therefore static, analysis for source-to-source transformations is the most extended for GPU devices. This is because the GPU yields great performance when tackling SIMD problems, such as for loops. An example of GPU static analysis can be seen in the work of Baskaran et al [17]. The PAR4ALL [99] and

PPCG [102] tools are also based on the polyhedral model.

Static analysis can be complemented by dynamic profiling. Static analysis has some weaknesses, mainly about its adaptability to changes in the hardware platform. Dynamic profiling can help cover these weaknesses.

Paralax [123] is a semi-automatic parallelization tool that relies on the assistance from the developers. Paralax leverages static and profiling information to construct an application dependence graph to perform dependence analysis. This tool detects pipeline-like parallelism. Another example of semi-automatic parallelization tools for a guided parallelism is the work proposed by Gohringer et al. [52]. This approach relies on a Polly extension and LLVM to automatically detect loops that significantly contribute to the overall application execution time.

The S2P tool presented by Athavale et al. [13] automatically parallelizes sequential C code. This tool detects source code regions by static and profiling analysis in order to transform it into OpenMP-like code. A similar approach is addressed by Jin et al. [71].

Codelet Extractor and REplayer (CERE) [29] is an open-source framework that finds and extracts the hotspots of applications, using a dynamic and static analysis approach. CERE isolates hotspots into code fragments, called *codelets*, in order to modify and measure them independently. This approach limits the analysis and modification of the original source code. However, once the source code is reduced to *codelets*, it is not possible to backtrace to the original version.

Lately, tools that work on a higher level have appeared. The idea is that, besides detecting pieces of parallel code, a deeper analysis is performed in order to detect high-level parallel patterns, of which the most extended seem to be the map and the pipeline patterns.

Nugteren et al. [94] combine the Bones source-to-source compiler [96] with the A-Darwin [95] extraction tool to transform C code into parallel skeletons on different computing languages, depending on the target device.

DiscoPoP [79, 61] leverages dependency graphs in order to detect parallel patterns like pipelines, using only profiling techniques. Nevertheless, this tool has two important drawbacks. First, the profiling techniques have a non-negligible execution time and memory usage. Second, they do not define any code assessment technique in order to translate the code for multiple accelerators.

A similar approach is presented by Tournavitis et al. [119], which detects and transforms legacy code into parallel code using parallel pipeline patterns.

The work presented by Grewe et al. [56] present an interesting twist with respect to the previous solution. The authors attempt to speed up OpenCL code in computing accelerators, but instead of starting from the raw sequential code, they use OpenMP code as a starting point. However, those techniques (based on decision trees) are not portable to new hardware accelerators and need a preliminary training phase before a regular execution.

Two major points are raised after studying these options regarding the strong

and weak points of static and dynamic code analysis. First, static analysis methods can only be used in simple scenarios, where pointer aliasing is not applied. This is because pointers may hide the actual memory address of a variable, or the elements of an array, which would make the study of data dependencies unfeasible at compile time. Second, dynamic analysis methods can cover most of the shortcomings of static analysis, but in return they can incur a major overhead in runtime that may affect the execution of the software proper.

## 2.5 Software partitioning techniques

The software partitioning techniques, in the context of heterogeneous parallel platforms, aim to assign different pieces of potentially parallel code to the available accelerators.

There is not a single or definitive way to classify the different partitioning algorithms. Rather, this Section will present a list of the different proposals that have been made over the years in order to enhance the performance of parallel applications.

The usage of OpenCL with CPU for HPC systems has been studied in recent years [107]. The conclusion is that the performance of OpenCL is close to OpenMP and library-based solutions, such as Intel Threading Building Blocks or Fastflow [5], with the advantage of multi-device code portability. Typically, OpenCL source code is fully portable, but lacks performance in some cases. Sangmin et al. [109] suggest some manual optimizations on the OpenCL code for a given benchmark in order to outperform its OpenMP counterpart.

Though until recently only CPUs and GPUs had available OpenCL drivers, different devices now provide OpenCL support. Altera's FPGAs [37, 10, 9], as well as Texas Instrument's DSPs [62] are some examples, though they provide older versions of the OpenCL standard and, even then, with limited support for OpenCL capabilities.

However, the host-device nature of OpenCL, coupled with its memory hierarchy model, results in a difficult choice whenever one needs to obtain as much performance from an heterogeneous parallel platform as possible. This problem becomes more pronounced as the amount of available accelerators increases.

Some works, therefore, propose to simplify the view that users have of the accelerators. One such work [72] suggests that all GPUs in the system can be unified under a single GPU device. In hiding the real architecture from the programmer under a unified view, programs written for a single GPU will work without needing to change the code. Internally, the algorithm splits single kernels between the available GPUs, so that each one computes part of the input data. The proposed runtime takes care of the memory management, so that any users don't have to worry about splitting the data or performing extra transfer operations. A major disadvantage of this solution is the exclusive work of GPUs.

A similar approach is that of the Load Balancing for OpenCL library (`lbcl`) [44]. This library creates a wrapper on top of the OpenCL library that splits kernels between the different available devices based on their computing power. It is important to note that, unlike the previous work, which completely hides the architecture, this solution actually provides a different API than OpenCL. While this means that it is impossible to simply reuse old OpenCL code without having to modify it, it also means that developing code from scratch with this library is easier, since it permits users to simply work with a high-level design.

The approach taken in `libWater` [54] is similar. The authors propose a higher level API than the one provided by OpenCL. This is an attempt to allow users to avoid the low-level implementation details. One of the differences is that `libWater` uses a Device Query Language that draws heavily from SQL to interact with the devices. However, rather than obscuring the underlying architecture, it relies on an event-based command scheduling.

The framework SkePu [45, 42] works with both multi-core and multi-GPU systems. In a similar fashion to the previous two works, it provides a high-level API based on the use of data-parallel and task-parallel skeletons. Its scheduling policy makes use of a mechanism called context-aware implementation selection. For this to work, different implementations for different devices must by available. SkePu will choose the fastest implementation at any time. It can also automatically split a single kernel between multiple GPUs.

There are several issues to point out from the previous works. First, OpenCL seems to be the unofficial standard used as a foundation layer for most of the higher-level solutions. Second, there seems to be a tendency to hide the actual architecture from the user. This liberates the user from having to waste time orchestrating tedious set-up and clean-up operations, and data-transfer operations, as well as having to worry about the best choice to run a parallel piece of code. Lastly, most of the solutions work at the kernel level. While operating at the kernel level has the advantage of allowing for fine-grained scheduling, such as splitting the computation of the kernel between different devices, it leaves little margin to overcome the data transfer overhead. Another interesting possibility, that may be combined with the kernel-splitting, is to schedule at the application level, considering the relations between different kernels, such as data dependencies. This would open the door to other possibilities, like using high-level parallel patterns, that only SkePu touches tangentially.

Another interesting point is that the partitioning is done automatically. Although with some frameworks or schedulers, users can manually specify their preferred partition, it is, once again, a tedious problem where the user needs specific knowledge about the architecture and the programming models in order to obtain a good performance.

Automatically choosing a partition scheme requires a previous analysis of the code. In this regard, there are two main approaches: static analysis and partitioning,

and dynamic profiling and partitioning. The static analysis techniques typically study the static qualities of the software, and decide a partition before execution. The information does not have to be necessarily the static code metrics, but it must decide on the partitioning before running the code.

The work from Grewe and O'Boyle [55] is one such algorithm. Here, the authors propose a static partitioning model based on characteristics extracted from OpenCL kernels (i.e. memory access patterns, McCabe's complexity number, number of numerical operations). This static partitioning model relies on a machine-learning approach to predict the best device for an OpenCL kernel. The model can decide to run a kernel in either the CPU or the GPU, or split the kernel in two parts to run them in parallel.

A more thorough solution is that of Albayrak et al. [2], that takes the profiling information for all kernels and performs a greedy search to find the best possible assignation. Unlike the previously explained work, this one does not consider splitting the kernel.

There are other semi-static approaches, like the one proposed by Kofler et al. [74], that have a static part and a dynamic part. This particular algorithm behaves similarly to the previous one, in the sense that the information is statically analysed, but then is partitioned dynamically. Other possibility is to generate ad-hoc code based on the static partition.

Other efforts statically combine partitioning based on both workload [111] and hardware capabilities [110], although predicting the input size before runtime can lead to poor results in certain scenarios, such as parallel functions inside library code.

In both cases, the proposed models are not easily scalable and adaptable when the heterogeneous architecture changes, because the predictive model needs to be trained again, increasing the system overhead. One example of dynamic overhead can be found in the Qilin compiler [81], which uses dynamic compilation in order to adapt to input memory size, and transferring the compilation overhead to runtime. In return, it can decide whether or not a GPU has enough memory to compute.

The tendency presented in the literature is to allow users to avoid taking low-level implementation decisions by automatically partitioning the code. In this line, there are two options: static partitioning, and dynamic partitioning. The static approach does not have an impact on the execution, but is also less flexible to changes in both the problem data size and the underlying hardware architecture.

## 2.6 Summary

In general, two main problems have been identified: adaptability to changes in the architecture and performance improvement. Regarding the first, the use of low-level solutions in all the researched areas often mean that changes in the architecture often affect negatively the performance. Also, by using low-level techniques, users

still require a certain amount of specific knowledge in order to make full use of the available technologies. Regarding the second problem, performance is seldom targeted in a specific manner. More often than not, the different technologies (i.e. code transformation, task partitioning) do not specifically target performance, and either work based on predictions or leave the optimization to the user.

This Thesis seeks to solve this problems by:

1. Using an architecture description of sufficiently wide coverage, and sufficiently detailed, that it can adapt to changes in the architecture and still provide enough information to allow for code optimization.

2. Using a high-level code annotation syntax that works as an abstraction from the underlying hardware. This

3. Developing a set of task partitioning techniques that specifically target a metric, in this case, code performance, by using empiric measures. These techniques will be able to adapt to changes in the architecture without the need to modify their implementation.

# Chapter 3

# A proposal for a new architecture description language

As seen in Figure 1.1, the underlying architecture will play an important role as an input for the partitioning techniques. A description of the architecture will be used to determine the availability, capabilities, and limitations of the different devices in the heterogeneous parallel platform. Additionally, both the annotation and partitioning techniques are designed around specific execution and hardware models. This chapter explains the basis of the work in future chapters. In Section 3.1 we discuss the designed execution and memory hierarchy models. Section 3.2 shows the architecture model used to describe heterogeneous parallel platforms. To recapitulate, this architecture description is one of the inputs for the partitioning techniques, as seen in Figure 1.1.

## 3.1 Programming model

The first proposal in this Thesis is to define a model that will be used to partition the software. This model must cover, at least, the execution of the different parallel tasks and the communication between the CPU and the accelerators. Therefore, we discuss the execution model and the memory model. Both are heavily based on OpenCL, which considers a master device (the CPU) and one or more slave accelerators, each with their own memory space.

### 3.1.1 Execution model

The assumed execution model is host-centric. Thus, the host device offloads user-defined *kernels* to target devices, or executes the kernels itself. *Kernels* are offloaded to the target device under control of the host. Even in device-targeted regions, the host may orchestrate the execution handling memory and data movements, queuing

the device code, and waiting for completion. In most cases, the host can enqueue a sequence of operations to be executed on the target device, one after the other.

By default, the model describes synchronous operations, where a host thread waits until the target device finishes the execution of a kernel. However, the model permits the asynchronous execution of kernels in target devices. Most of the target devices can manage executions from the host thread asynchronously. Such target devices have one or more activity queues. The host thread will enqueue operations onto the device activity queue, such as data transfers and procedure execution. After enqueuing the operation, the host thread can continue the execution while the device operates independently and asynchronously. The host thread can query the device activity queue and wait for its operations to be completed. Operations on a single device activity queue will be completed before starting the next operation on the same queue; operations on different activity queues may be active simultaneously and may complete in any order. Each target device has its own execution threads and resources. A kernel running on one target device cannot be migrated to another target device.

The model guarantees that input and output kernel parameters are syncronized at the beginning and at the end of the target device execution. There is no data synchronization between asynchronous kernel executions.

If a target device does not exist, the target device is not supported by the implementation, or the target device cannot execute the target construct then the target region is executed in the host device. The target device execution model depends on its family and its hardware architecture (e.g. GPU, FPGA).

Figure 3.1 shows the execution model workflow for running a computational kernel in an accelerator.

The programmer has limited control over the decision taken by the framework in terms of the kernel execution. The programmer can select the type of the target device family for kernel regions but not the specific device of that family. For example, the programmer could decide to use a kernel only in a GPU device, but in a heterogeneous parallel platform with more than one, the scheduler will decide the best one where the kernel will be executed. The reason for this is tied to the annotation mechanisms described in Chapter 4, which are designed to be a high-level asset for programmers.

### 3.1.2 Memory model

The proposed memory model considers separate memory spaces for the host and the target devices. It is necessary to use data transfer mechanism between the host and a target device. In the defined programming model, data movement between memory spaces (host and a target device) will be managed by a source-to-source transformation process, based on the kernel parameter attributes defined by the programmer and the framework to be used in the final source code. Figure 3.2

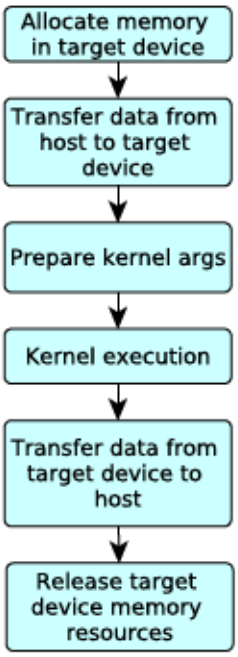


Figure 3.1: Proposed execution model.

shows the memory model scheme.

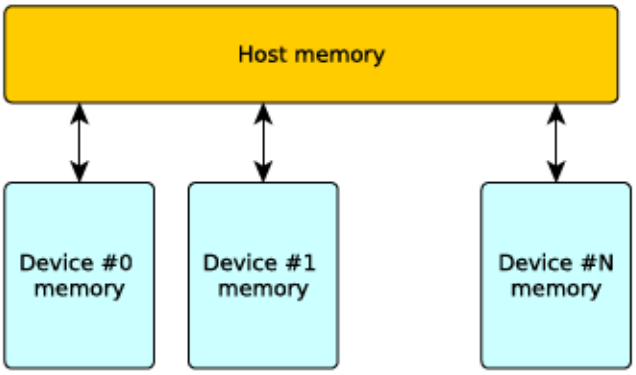


Figure 3.2: Proposed memory model.

Having separate memory spaces for the host and the devices has some implications for users, including but not limited to:

- **Memory bandwidth** between host memory and target memory determines the level of computation intensity required to effectively accelerate a given kernel.

- The limited **device memory size** may prohibit offloading of kernels that operate on very large amounts of data.

- **Host addresses** stored in pointers on the host may only be valid on the host; addresses stored in pointers on the device may only be valid on the device. Dereferencing host pointers on the device or dereferencing device pointers on the host is likely to be invalid on such targets.

Target device data have an explicit lifetime, from when it is allocated until it is released. The memory model defines different lifetime of the kernel variables depending on the scope of the kernel:

- **Parameter**: It is a host variable used in the kernel code. If this variable is a kernel input, it must be stored in the target device memory before kernel execution. If this variable is a kernel output, it must be send to host after kernel execution.

- **Local**: Variables defined inside the kernel region and not used outsize of it. In this case, the local variables are created, managed, and destroyed by the target device.

## 3.2 A new architecture description language

The objective of this Thesis is to run parallel code in heterogeneous parallel platforms of diverse characteristics and capabilities. The advantage of such platforms with different processors is that they are adaptable and provide the possibility to optimize different applications. Therefore, it does not make sense to limit this study to a single, fixed platform. In order to work with dynamic platforms, it is necessary to learn their composition on-the-fly. To this end, we have developed a hardware description language, which is based on the execution model explained before. For the sake of feasibility, the scope of the model will be the following processors: CPU, GPU, FPGA and DSP.

### 3.2.1 Hardware parallel platform

The Heterogeneous Parallel Platform Description Language (HPP-DL) is a specification of a human-readable language that provides all the relevant details of a heterogeneous parallel platform. Heterogeneous platforms can be made of multicore, GPU, FPGA, DSP, or combinations of all the previous. HPP-DL gathers the architectural information used by heterogeneous computing systems and defines architecture patterns. HPP-DL is designed to be human readable, so that automated and non-automated descriptions of platforms can be made.

JSON (JavaScript Object Notation) [36] format will be used to represent the HPP-DL information. The main motivations to adopt JSON as, representation language of HPP-DL are: 1) J,SON is a lightweight data-interchange format; 2) it is easy for humans to read and write; 3) it is easy for machines to parse and generate; and 4) it is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd edition - June 2011 [50]

HPP-DL allows to express the characteristics of a hardware system via a hierarchical representation. Its purpose is to ensure that platform-specific information is made available to expert programmers and tools such as auto-tuners, compilers, and run-time systems. The HPP-DL format is independent of the programming model employed. This means that it can be used as a virtual platform for offline hardware simulators.

HPP-DL is based on other previous works, such as the *Hardware Lister (lshw) [46]* and *Hardware Locality (hwloc)[116]* tools. One of the weaknesses of *lshw* is that, although detailed, it only works on CPUs. On the other hand, *hwloc* describes the whole architecture from a hierarchical point of view, but is not very detailed about the characteristics of each component. HPP-DL is a detailed language which can be used to describe a wide array of components.

The abstract entity described by the HPP-DL is known as the Hawdware Parallel Platform (HPP). It contains all the elements needed to describe a heterogeneous platform:

- The hardware elements that make up the platform, `components`. Examples of components are the devices and the memory modules.
- The connections between them, `links`. The PCIe buses are one example of links, as are the connections between cache levels.
- The data shared by these elements, `resources`, such as I/O ports in an FPGA.

These elements are nested in a hierarchical structure, where each level is part of the previous one. An example can be seen in Figure 3.3. In the figure, we present a platform with a CPU and two GPUs plus the main memory. The CPU has a core, with a local cache. There is also a PCI link that is used to connect to the GPUs, plus some additional information, such as available I/O ports in an FPGA, which are represented as memory addresses.

The three elements, components, links, and resources, are described in the following subsections. A formal definition of the language can be found in Appendix A. Here, the rules to generate all the elements described further are shown, along with several examples. A full description of the reference platform used in the tests is also included, and Listing 3.1 shows a simplified description of an heterogeneous parallel platform.

Listing 3.1: Small example of HPP file

```
{
  /* Metainformation of HPP-DL */
  "class": "hpp",
  "description": "Human readable description",
  "version": "1.0",
  "date": "2014-01-13 10:00",
  "components":[
    /* Definition of harware platform */
    {
     "class": "platform",
     "id": "platform:0",
     "description": "REPARA Reference System. X9DRG-QF (To be filled by O.E.M.)",
     ...
    },
    /* Definition of processor 0 */
    {
      "class": "processor",
      "id": "platform:0.processor:0",
      "description": "Intel (R) Xeon(R) CPU E5-2620 0 @ 2.00GHz",
      ...
    },
        ...
        /* Definition of GPGPU */
    {
        /* Common attributes */
        "class": "gpgpu",
        "id": "platform:0.gpgpu:0",
        "description": "geforce gtx titan",
        ...
    },
        ...
   ],
    /* Definition of links */
  "links":[
    /* Definition of gpu -> pci link */
    {
      "class": "link",
      "id": "link:0",
      "description" : "Link between gpgpu 0 and pcie 0",
      "src_component": "platform:0.gpgpu:0",
      "dst_component":"platform:0.pcie:0",
      "throughput" : [
         {"size":1, "value":"1 KB/s"},
         {"size":2, "value":"2 KB/s"},
         /* rest of measures */
      ],
      "latency": "0.8 ms"
    },...
  ],
  /* Definition of memory resources */
  "resources":[
    {
      "class": "resource",
      "id": "resource:0",
      "description" : "",
      "component_ref": "platform:0.memory:0",
      "io_memory_address": [ "0x200000000-0x24000000" ],
      "io_ports": [ "0xdc00-0xdf00", "0xfe00-0xff00"],
      "irq": "91"
    },
        ...
  ]
}
```

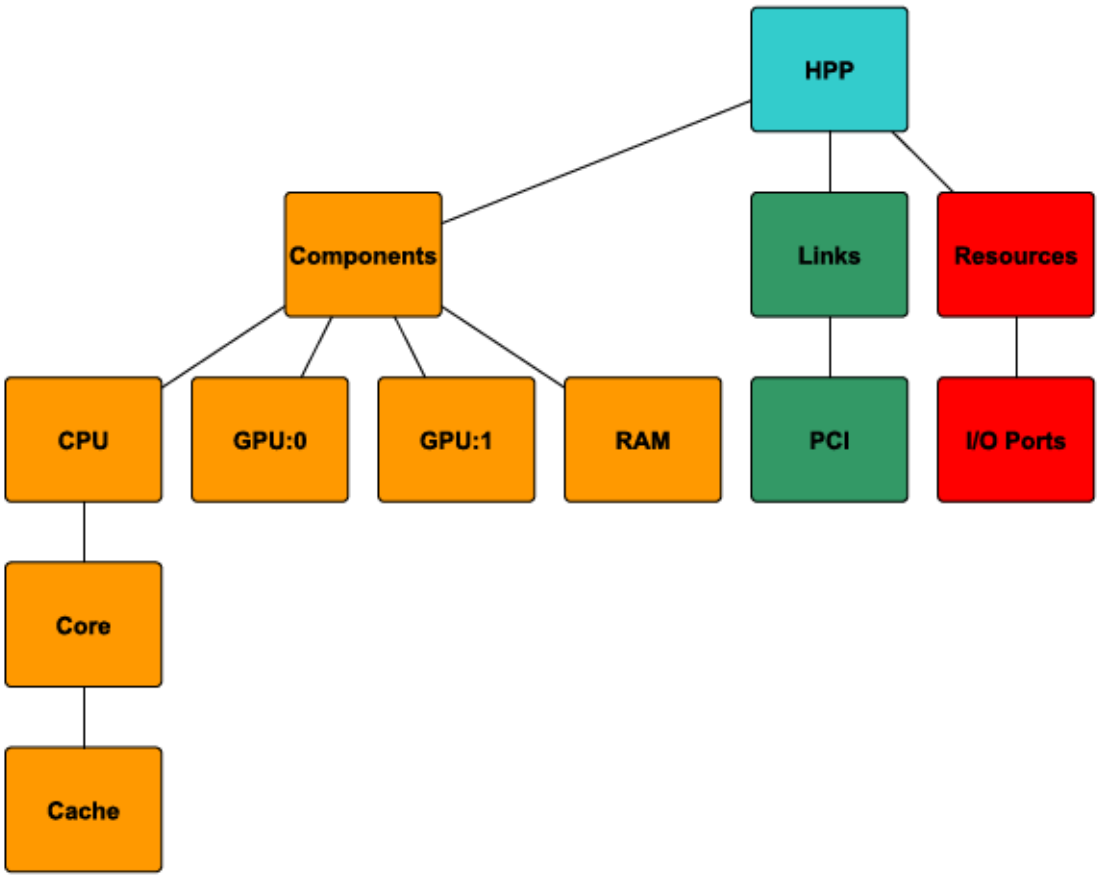


Figure 3.3: Example of HPP hierarchy.

### Components

A component is one of the hardware elements that make up the whole HPP. Components are connected to other components in various ways, forming the heterogeneous platform. Different components have different attributes, depending on their nature. For the sake of feasibility, the following components are considered:

A **platform** represents the motherboard where application programs are executed. It is composed by hardware components, forming a heterogeneous hardware platform. It holds many of the HPP-DL components: main memory, memory banks, processors, etc. Components are connected with the platform through PCIe component (i.e. GPU). In the current version of HPP-DL specification, only one platform per HPP-DL description is allowed.

The **memory** is an abstraction that represents the available memory in the platform or a component. It is formed by memory banks.

In HPP-DL it is possible to define two different CPU memory models:

- Symmetric MultiProcessing (SMP)

- Non-Uniform Memory Access (NUMA)

**Symmetric multiprocessing (SMP)** is a multiprocessing architecture in which multiple identical CPUs (multicore), residing in one cabinet, connected to a single shared main memory. In HPP-DL, a SMP architecture is represented connecting all memory banks to all processors using $N$ link objects where `throughput` and `latency` attributes contain the same values.

**Non-Uniform Memory Access (NUMA)** is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). In HPP-DL, a NUMA architecture is represented connecting all memory banks to all processors using $N$ link objects where `throughput` and `latency` attributes contain the different values. Thus, the latency will be lower and the throughput will be higher when the processor and the memory bank are closer.

Figures 3.4 and 3.5 show examples of SMP and NUMA architectures, respectively.

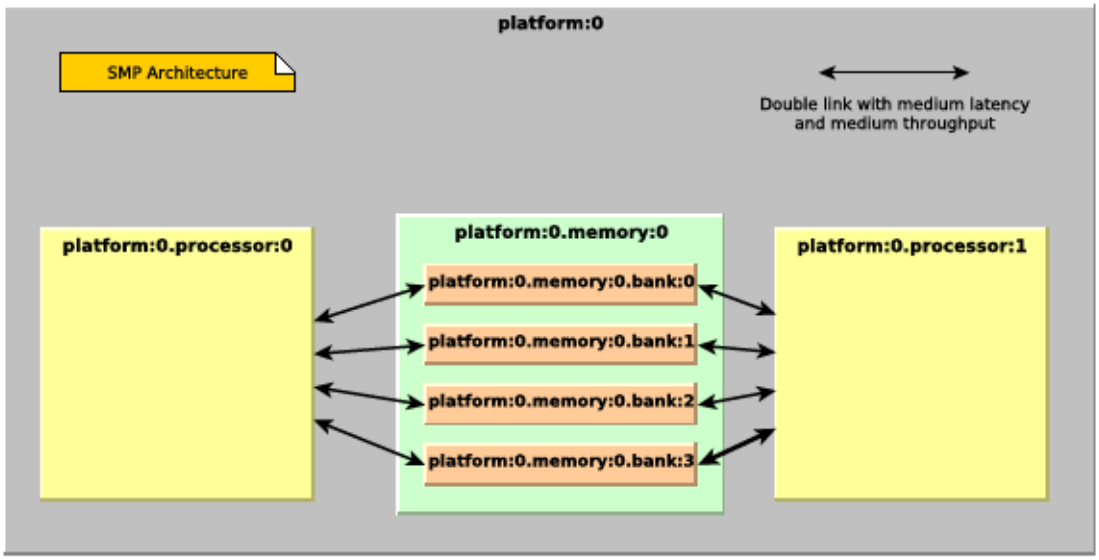


Figure 3.4: Scheme of SMP architecture on HPP-DL.

The **memory banks** represent a physical memory bank inserted into a slot. In HPP-DL, memory bank includes the hardware characteristics. A memory bank belongs to a memory of a component.

The **processor** component. The platform can contain multiple CPUs with multiple cores. These CPUs have different architectures depending on the design memory access.

**Core**. Integrated single processor on a more general processor, where other cores are attached. Core is connected with cache components (i.e. instruction or data caches) using link object. In HPP-DL, a core is a processing unit included on a computing device such as CPU, or DSP. Usually, these elements are grouped into processor components, sharing resources as caches or memory. Three families of cores are considered in HPP-DL specification: CPU, DSP and FPGA. The hardware architecture of the core determines its family.

**Cache**. In HPP-DL, the cache component describes the hardware characteristics of the cache memory banks. Typically, a cache is connected with cores, caches or other processor component by using link objects.

In HPP-DL, a **GPU** is a device that increases the performance of the software applications. Multiple heterogeneous architectures are used as GPUs. To unify all possible GPUs devices, we have used the OpenCL 2.0 specification [98] as standard representation. OpenCL 2.0 defines an enhanced execution model and a subset of the C11 and C++11 memory model, synchronization and atomic operations. This class only considers OpenCL devices that have the `CL_DEVICE_TYPE` attribute with a value equal to
`CL_DEVICE_TYPE_GPU`. HPP-DL contemplates two different cases: GPU boards and integrated GPUs. GPU boards are devices connected to a processor through a PCIe interface. Integrated GPUs are part of a processor component, and can be linked to them through their unique id. Examples of the latter can be seen in the examples section of the PCI component Annex A.1.11.

The **PCIe** component is used to interconnect components located on a platform or on boards attached to this platform (e.g. GPU board, FPGA board, etc.). In the

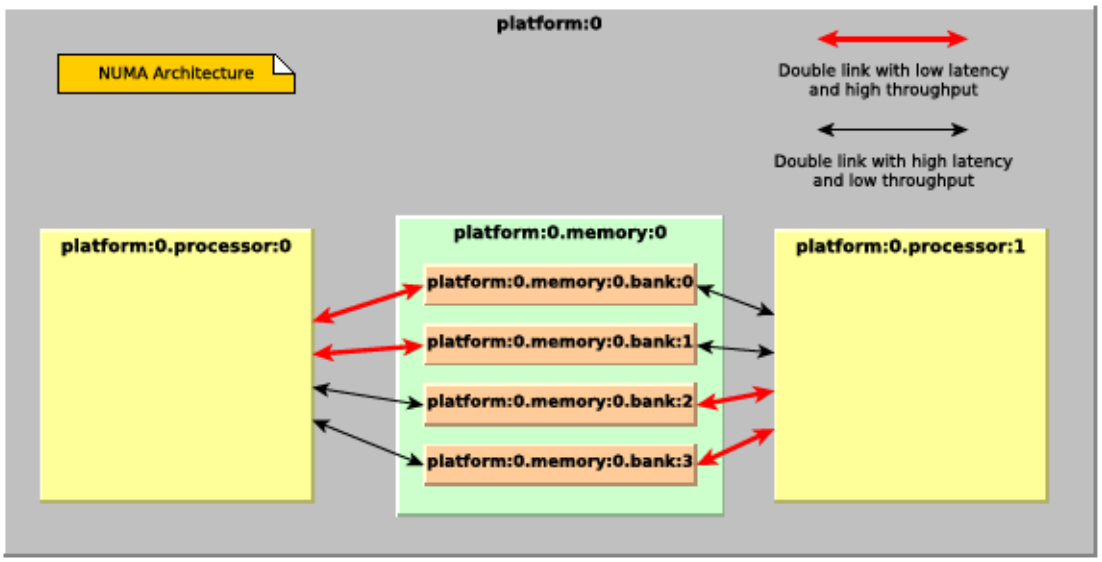


Figure 3.5: Scheme of NUMA architecture on HPP-DL.

current version of the HPP-DL specification, PCIe is the only interface described. PCI Express is a high performance standard type of connection, for internal platform components. It defined as a general purpose serial I/O interconnection for a wide variety of future computing and communication boards/platforms. In HPP-DL specification, PCIe component refers to the actual expansion slot on the base platform that accepts PCIe expansion cards such as GPU or FPGA boards. PCIe component defines the characteristics of this physical slot for component interconnection. Always It belongs to the platform component, including platform as parent component of PCIe ID component (i.e. `platform:0.pcie:0`). The rest of the components could connected with PCIe component by using link objects that include the transfer characteristics of the connection.

**FPGA board**. FPGA devices appear in different forms and architectures, from pure accelerator cards with on-board memory (similar in design to GPUs) to System-on-Chip (SoC) approaches where the reconfigurable fabric is tightly connected with one or more general purpose CPUs capable of running a fully featured modern operating system, such as Linux. In HPP-DL FPGA devices are modelled by instances of the `FPGA` class, which inherits from the `Component` class: An instance of the `FPGA` class shall represent the device as a whole, e.g. a PCI board from a specific vendor, such as the VC709 from Xilinx. The `FPGACore` class can model the actual FGPA chip used on the board. As with the other devices described in previous sections, additional components such as memory elements (caches, RAM, ...) and connectivity information (e.g. PCI slot specification) may be specified to describe the components of the device at a chosen level of detail. FPGA chips are architecturally and structurally so different from each other that it makes little sense to model them on a finer level in HPP-DL. Furthermore the exact chip model (in most cases even only the board identifier) is usually sufficient for synthesis and bitstream generation with the vendor's toolchain.

**DSP board**. This component is based on PCIe DSP boards DSPC-8681 and DSPC-8682 [1]. A DSP board is a computation device that is subordinated to a HPP-DL platform composed by several processors with several DSP cores. It integrates one or several DSP processors (see Section A.1.4) where there are several DSP cores inside, common memory, and memory banks. It is connected with the platform by using a communication interface (PCIe) by using a link entity. It can include a FPGA core for PCIe communication, but it is irrelevant to computing. These processors do not include GPU components.

### Links

This HPP-DL entity represents the relationships between two different components included on HPP-DL specification without a membership relation. The relationship is one-way direction; where there are two different points defining the link: source and destination. A bidirectional communication is defined with two link objects, where the source and the destination are exchanged. The link includes the infor-

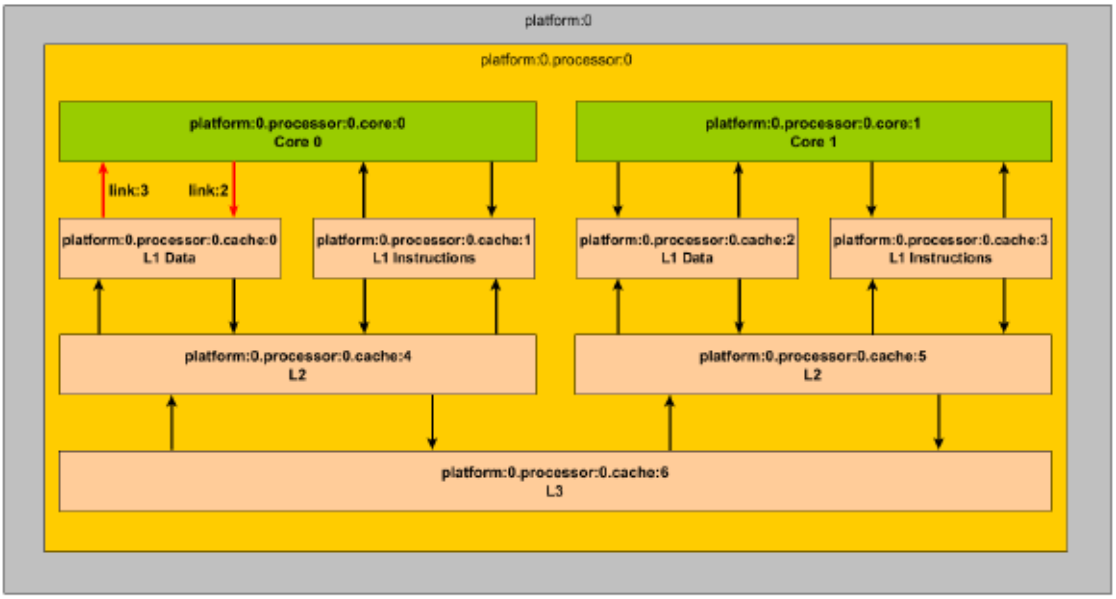


Figure 3.6: Link scheme of interconnections between cores and caches inside a processor.

mation about the characteristics of the data transmission: throughput and latency. Figure 3.6 shows an example of the links between cores an cache memories inside a processor.

Throughput is the effective bandwidth obtained transmitting on the link a specific data size; meanwhile latency is the amount of time that data takes to traverse a connection. These values show the real restrictions of the link connection to send data between to components. In some cases, throughput and latency could have the default values to represent just a relationship, not a connection (e.g. a core and an associate cache).

It is for the previous reasons that boards connected through PCIe and other components are related to a platform through a link, rather than belonging. Since a board's effective bandwidth and throughput depend on both the board and the PCIe link, it is better to establish them in a separate entity altogether. Additionally, this allows covering the possibility of a two-way channel with different values for each direction.

### Resources

HPP-DL resource contains OS-specific information about resources used by / allocated to a component, such as I/O ports, IRQs or address ranges. It is used mainly to develop code with FPGA boards, where low-level memory operations are needed.

## 3.3 Summary

In this chapter we have proposed an architecture description language, as well as a programming model that defines the behaviour of the different devices in a heterogeneous parallel platform. These two contributions will be the basis for the rest of the proposals in the Thesis. In particular, the programming model affects the definition of the software annotation and the task partitioning techniques. The model is based on OpenCL, and defines a master-slave relationship between the CPU and the accelerators, where each one has its own memory space. On the other hand, the architecture definition is a necessary input for the task partitioning techniques. In the following chapter, we will show the proposal for the other building block: the annotations and the automatic annotation techniques.

# Chapter 4

# Kernel identification and code annotation

As seen in the previous chapter, the chosen model is based on OpenCL. The general idea is that a host will orchestrate the execution of parallel pieces of code, or kernels, amongst all the available processors, including the host. Each device has its own separate memory space, and part of the partitioning process will include handling date movement.

The necessary information will include, but not be limited to, identifying kernels, identifying input and output variables, and associating kernels to a specific parallel pattern (e.g., map, farm, pipeline).

This information must be introduced inside the code, either manually or via and automatic process. The information must be a high-level abstraction, simple to use, and it must ensure the code maintainability, in case users decide to do the partitioning process manually.

In this chapter, we propose a software annotation syntax based on the restrictions derived from the programming model, as well as a set of tool that automatically detect and annotate kernels.

In Section 4.1 we describe the annotation format specification. Section 4.2 shows the automatic kernel identification and annotation techniques.

## 4.1 Software code annotation specification

In this section we describe the proposed specification for the code annotations. They are designed to allow users to specify the behaviour of the code from a high-level perspective.

Listing 4.1: Example of a kernel annotation.

```
1  /* Sample of a kernel that includes the basic attributes *
2   * defined on the specification:                         *
3   * - target where the code could be executed,            *
4   * - input variables                                     *
5   * - and output variables                                *
6   * The variables defined on the kernel are privated.     */
7  [[rpr::kernel,
8    rpr::target(CPU, GPU),
9    rpr::in(A,B,n,data),
10   rpr::out(C)                          ]]
11 for (int i = 0; i < n; ++i)
12   C[i] = A[i] * B[i] + data ;
```

### 4.1.1 Annotation format

The syntax for the source code annotations is based on the C++11 attributes defined in the C++11 standard [70]. The particular syntax used for the statements is the following:

[[ *rpr-attribute-list* ]]
`kernel-region`

where a kernel region is a statement or a compound statement.

In order to distinguish our attributes from the rest of the C++ attributes, the first will start with the namespace *rpr*, followed by the name of the attribute. Furthermore, all the annotations may have one or more attributes, separated by commas. Listing 4.1 is an example of an annotated kernel.

Attributes will be split in four categories, depending on their target area, like data-related attributes or high-level design attributes. Table 4.1 summarizes the proposed C++ attributes.

- **Core attributes**. These attributes are used to mark kernels as well as their desired behaviour. They can be used to indicate compatible target processors, as well as whether the kernel will be run asynchronously or not.

- **Data-related attributes**. These attributes are used to provide information on the transference of the variables used inside a kernel from the host to the device and viceversa. They are useful for detecting dependencies, as well as to optimize transfers between the host and the devices.

- **High-level parallel patterns**. These attributes allow to indicate if a kernel, or a set of kernels match a particular parallel pattern. They are useful to optimize code, based on the final generated code.

- **Auxiliary attributes**. These attributes are not meant to be used with a release version of the parallel code, but rather to provide information during testing.

Table 4.1: Proposed C++ attributes.

| **Type** | *Name* | **Description** |
|---|---|---|
| Core | `rpr::kernel` | Determines parallel region in a source code. |
| | `rpr::target` | Determines the target device family for a kernel. |
| | `rpr::async` | Defines if a kernel is executed asynchronously. |
| | `rpr::sync` | Allows synchronizing asynchronous kernels. |
| Data-related | `rpr::in` | Defines kernel input parameters. |
| | `rpr::out` | Defines kernel output parameters. |
| | `rpr::keep` | Keeps data in the internal memory of a device, avoiding transfers between host-devices. |
| | `rpr::size` | Determines the problem size of a kernel. |
| | `rpr::pattern` | Defines the access pattern of an array/vector. |
| | `rpr::reduce` | Determines kernel variables that are defined by a reduce operation. |
| | `rpr::stream` | Determines which variables will be part of a data stream in a pipeline computation. |
| High-level parallel patterns | `rpr::pipeline` | Defines pipeline parallel skeleton in source code. It is followed by the `stream` attribute. |
| | `rpr::farm` | Specifies the farm skeleton as execution model in a kernel. |
| | `rpr::map` | Determines the map skeleton as execution model in a kernel. |
| Utility | `rpr::log` | Enables profiling information from a defined kernel. |

### 4.1.2 Core attributes

These attributes are used to mark kernels as well as general information pertaining the behaviour of the kernel. This information includes preferred devices where the kernel can be run, as well as whether or not the master device can run the kernel in an asynchronous manner.

#### rpr::kernel

The *kernel* attribute is applied by the programmer before a single or compund statement to mark them as a kernel region.

When the partitioning algorithm finds a code block marked as a `kernel`, it analyses the kernel to find any characteristic relevant to the partitioning process, and then decides a target device for the `kernel`. This is the most basic attribute in the `kernel` definition language. The rest of the attributes cannot be applied unless they are related to a *kernel* attribute.

The syntax of *rpr::kernel* attribute is as follows:

[[ **rpr::kernel**, *rpr-attribute-list* ]]
`kernel region`

It can be used with: 1) single or multiple statements involved in a compound statement, 2) loops, and 3) function/member function calls.

I/O operations can be considered as kernels when streams are used (See *rpr::pipeline* below). In this case, a kernel will be defined as a stage of the pipeline.

The *rpr::kernel*:

- can only be included once per kernel attribute definition.
- cannot be used to nest kernels. If two or more different kernels are nested, only the outermost will be taken into account.

Listing 4.2 is an example where a kernel is defined.

#### rpr::target

Attribute that allows the programmer to indicate the type of device to run a kernel. Barring some restrictions, such as the use of dynamic memory, a kernel can run in any processor. However, as has been explained, some devices are better suited to run software with certain characteristics. This **rpr::target** attribute allows to limit the possible processors if the user or a static analysis want to indicate so. The *rpr::target* attribute is mandatory.

The syntax of *rpr::target* attribute is as follows:

Listing 4.2: `kernel` attribute example.

```
1  /* Different examples on the use of the kernel clause */
2
3  /* for loop as a kernel */
4  [[ rpr::kernel,
5          ... /* rest of attributes */ ]]
6  for(i=0; i<N; ++i) {
7          ...
8  }
9
10 /* compound statement as a kernel */
11 [[ rpr::kernel,
12    ... /* rest of attributes */ ]] {
13         ...
14         A = ...
15         ... = B
16 }
17
18 /* expression as a kernel */
19 [[ rpr::kernel,
20         ... /* rest of attributes */ ]]
21 ret = perform_rotation(...);
```

[[ **rpr::kernel, rpr::target(** `target-device`, *rpr-target-device-list* **),** *rpr-attribute-list* ]]

`kernel region`

*rpr::target* mandatory attribute is applied with a kernel.

The list of desired target devices will be indicated between parentheses as a comma separated the attribute parameters. This list cannot be left empty.

The `target-device` parameter must be one of the following:

- `CPU` if the kernel may be run in a CPU.
- `GPGPU` if the kernel may be run in a GPGPU.
- `FPGA` if the kernel may be run in a FPGA.
- `DSP` if the kernel may be run in a DSP.
- `ANY` if the kernel may be run in any kind of device. It is incompatible with the rest of the `target-device` values.

The *rpr::target* attribute:

- is mandatory for kernel region definition.

Listing 4.3: `target` attribute example.

```
/* The kernel will be executed on GPGPU components      *
 * of the system. The particular device will be         *
 * chosen by the scheduler.                             */
[[ rpr::kernel, rpr::target(GPGPU),
... /* rest of attributes */ ]]
for(auto i=0; i<N; ++i) {
        ...
}

/* The kernel could be executed on CPU or FPGA device   */
[[ rpr::kernel, rpr::target(CPU, FPGA),
... /* rest of attributes */ ]]
for(auto j=0; j<N; ++j) {
        ...
}

/* The kernel could be executed on any device           */
[[ rpr::kernel, rpr::target(ANY),
... /* rest of attributes */ ]]
for(auto i=0; i<M; ++i) {
        ...
}
```

- can be used to indicate a target device which is not really part of the hardware platform [104].

Listing 4.3 is an example of three different uses for the *target*, one with only one target, another with different targets for a kernel, and finally one that could be executed on any kind of computing element of the hardware platform.

#### rpr::async

Defines attributes that allow asynchronous kernel execution. Asynchronous kernels are executed in parallel with the host program execution. When running a synchronous kernel, the scheduler assumes that the host program will perform setup operations, execute the kernel, then wait for its completion. The `rpr::async` attribute indicates that the master program will skip the last step. Several consecutive kernels may be defined as asynchronous, being executed independently between them. In all cases, the master program will wait for the first synchronization point before performing any cleanup operations or data transfers after the asynchronous kernels have finished.

The syntax of *rpr::async* attribute is as follows:

Listing 4.4: async attribute example where a final kernel waits the async kernels

```
1  /* Asynchronous kernel */
2  [[ rpr::kernel, rpr::async, ... ]]
3  for(i=0; i<N; ++i) {
4          ...
5  }
6  /* sequential code */
7  ...
8
9  [[ rpr::kernel, rpr::async, ... ]]
10 for(i=0; i<M; ++i) {
11         ...
12 }
13 /* sequential code */
14 ...
15
16 /* Kernel that syncs the previous kernels    *
17  * before the execution of this one.         *
18  * This means: "wait for all async kernels" */
19 [[ rpr::kernel, ... ]]
20 ret = perform_rotation(...);
```

[[ **rpr::kernel, rpr::async** , *rpr-attribute-list* ]]
{*kernel region*}

A synchronization point can be:

- before the execution of a synchronous kernel.
- before the execution of statement marked with *rpr::sync*)
- at the end of a compound statement where asynchronous kernels were defined.

The asynchronous kernels do not have any consistence memory mechanism. It is the responsibility of developers not to modify shared data between these asynchronous kernels.

The *rpr::async* attribute:

- can only be included once per kernel attribute definition.
- can only appear in the definition of a kernel.

Listing 4.4 is an example where there are some async kernels executed in parallel. In this case, a final synchronous kernel waits the results of the previous async kernels.

Listing 4.5: Example of synchronization of asynchronous kernels defined in a function.

```
/* Other sample of async kernels in a code       *
 * region.                                        */
void foo(....){
  /* Asynchronous kernel 1 declaration */
  [[ rpr::kernel, rpr::async, ... ]]
  for(j=0; j<M; ++j) {
        ...
  }
  /* sequential code */
  ...
  /* Asynchronous kernel 2 declaration */
  [[ rpr::kernel, rpr::async, ... ]]
  for(i=0; i<N; ++i) {
        ...
  }
}/* Synchonization point */
```

Listing 4.6: Example of synchronization of asynchronous kernels defined in a nested loop.

```
/* Asynchronous kernels declared inside a  *
 * structure code block (e.g. body loop)   */
for (i=0; i<N; ++i) {
  /* Asynchronous kernel declaration */
  [[ rpr::kernel, rpr::async, ... ]]
  for(j=i; j<M; ++j) {
        ...
  }
  /* sequential code */
  ...
}
/* There is an implicit sync operation, where it *
 * waits that all async kernels finalize.        */
```

Listings 4.5 and 4.6 show examples where there are some asynchronous kernels are executed in a statement. In this case, the synchronous operations are done after the statement ends its execution.

#### rpr::sync

This attribute allows waiting for previous asynchronous kernel executions, continuing the execution when all asynchronous kernels have finished. It is not associated with any kernel definition.

Listing 4.7: sync attribute example.

```
1  /* Asynchronous kernel 1 */
2  [[ rpr::kernel, rpr::async, ... ]]
3  for(i=0; i<N; ++i) {
4          ...
5  }
6  /* sequential code */
7  ...
8
9  /* Asynchronous kernel 2 */
10 [[ rpr::kernel, rpr::async, ... ]]
11 for(i=0; i<M; ++i) {
12         ...
13 }
14 ...
15 /* sync attribute works as a barrier */
16 [[ rpr::sync]] f();
```

The syntax of the *rpr::sync* attribute is as follows:

[[ **rpr::sync** ]] `statement`;

The *rpr::sync* is ignored if there are no previous asynchronous kernels.

Listing 4.7 is an example where there are some asynchronous kernels are executed in parallel, and one `sync` operation is done at the end.

### 4.1.3 Data-related attributes

These attributes pertain to the behaviour of the data in the memory of the host or the accelerators. With these attributes, users can indicate whether a variable is part of the input or the output of a kernel, or whether or not to release the allocated memory after a kernel is finished. They can also be used to give useful information to optimize the task partitioning process, such as maximum expected size or if a variable is accessed according to a specific memory access pattern.

#### rpr::in

The *rpr::in* attribute identifies which variables constitute the input parameters for the kernel execution.

The selected *in* variables must be stored in target device memory space before kernel execution. If any of these variables are not in target device memory space, they must be transferred from host to target device. The host is in charge of storing

the input parameters before the kernel execution. The transference to the target device could be done using several ways, normally by copying the variable data to another variable stored on the target device memory space but also by making the variable accessible to the target device from the host memory space.

It is possible to use the *rpr::in* attribute over a delimited range of an array variable. The Intel array notation is used to define the range. This notation allows to define, for each dimension, the lower bound and the length of the range. The stride can be defined too, to allow ranges with a simple stride pattern. Thus, only the data on this array range will be accessible from the target device and will be marked as constant.

The variables marked as *in* that are not defined as *out* parameter in a kernel, are considered as non-modified during kernel execution. This means that, after the kernel has been executed, the value of the host input variable is the same as before the execution. The whole value of the variable is considered non-modified, no matter how many pointers or references exist in the middle. Otherwise, if *in* is modified inside a kernel, it raises an undefined behaviour.

The syntax of the *rpr::in* attribute is as follows:

[[ **rpr::kernel, rpr::in(** `variable` *[array-notation] , var-in-list* **)** *, rpr-attribute-list* ]]
`kernel region`

*rpr::in* attribute is applied to a kernel (See Section 4.1.2) and it is mandatory if there is at least one input. The arguments of the attribute are the list of variables, separated by commas, that act as input parameters of the kernel.

If the input variable is an array, then the `Intel array notation` [66] must be used to define the range of the array that constitutes the input argument. This notation allows to define, for each dimension, the lower bound and the length of the range. The stride can be defined too, to allow ranges with a simple stride pattern. It will be defined by specifying per each dimension of the array. It is possible to extend this notation to `standard template containers` (e.g. `std::vector`) to define the size of the input parameter.

Regarding the use of the *rpr::in* attribute:

- can only be included once per kernel attribute definition.
- can only appear in the definition of a kernel.If it is not included in kernel definition, the kernel has no input parameters.
- a variable defined as an *in* parameter may also be defined as an be also *out* parameter.

Listing 4.8: `in` attribute example.

```
1  /* A is a input kernel value       */
2  int A[1000];
3  /* B is a input kernel value, but *
4   * only from indices 200 to 300     */
5  int B[1000];
6  /* C is a multiarray input kernel */
7  int C[100][100];
8
9  [[ rpr::kernel, \
10    rpr::in(A[1000],B[200:100],C[100][100]),
11    ... /* rest of attributes */ ]]
12 for(i=0; i<1000; ++i) {
13     ...
14     j = A[i];
15     k = B[200 + (i % 100)];
16     k += C[i][i]
17     ...
18 }
```

- can be used in combination of *rpr::keep* attribute where a *in* variable may be kept in memory device after kernel execution.

Listing 4.8 is an example of an array of 1000 integers (`A`) used in its entirety as an input parameter for the selected kernel and another array of 1000 integers (`B`) where only a range of 100 is used, starting from element 200. Finally, a multi-array (`C`) with 100x100 elements is defined as input parameter.

The kernel implements a loop where each iteration reads one of the values of the input arrays.

### rpr::out

*rpr::out* attribute identifies which variables constitute the output parameters for the kernel execution.

At the end of kernel execution, the selected *out* variables may be stored:

- In the same target device memory, if the following kernels use these variables as *input*.
- In the host memory for other cases. The kernel output data are transferred to the host on these variables at the end of the kernel execution. The transfer to the target device may be done using several ways. Normally the host copies the output data from the corresponding variable stored in the target device memory space at the end of the kernel execution. Also the host can make the variable accessible from the target device. In this case, the target device

is the one that copies the output data onto these variables during the kernel execution.

A variable can be defined as *output* only if it does not appear in the *input* kernel list. In this case, the semantic of this attribute enforces that the initial value of these variables is unused by the kernel, so no assumptions can be made about their initial value. The output value of the host variables is set only after the kernel is executed. Nothing can be assumed about the host variable values during the kernel execution.

The syntax of *rpr::out* attribute is as follows:

[[ **rpr::kernel, rpr::out(** `variable` *[array-notation] , var-out-list* **)** *, rpr-attribute-list* ]]
`kernel region`

*rpr::out* attribute is applied to a kernel and it is mandatory if there is at least one output parameter. The arguments of the attribute are the list of variables, separated by commas, that act as output parameters of the kernel.

*rpr::out* variables uses the same rules of construction than *rpr::in*.

The *rpr::out* parameter:

- can only be included once per kernel attribute definition.
- can only appear in the definition of a kernel. If it is not included in kernel definition, the kernel has no output parameters.
- can be used in combination of *rpr::keep* attribute where a *out* variable may be kept in memory device after kernel execution, avoiding transfer it at the end of kernel execution. The developer should avoid *memory consistency problems*.

Listing 4.9 is an example of an array of 1000 integers (J) used completely as an input parameter for the selected kernel and another array of 1000 integers (K) where only a range of 100 is used, starting from element 200. Finally, a multi-array (L) with 100x100 elements is defined as output parameter.

The kernel implements a loop where each iteration writes down one of the values of the output arrays.

### rpr::keep

The *rpr::keep* attribute gives the hint to keep one or several variables between kernel executions, by avoiding delete or release operations at the end of kernel execution. This attribute must be used with variables that are not modified between kernels

Listing 4.9: out attribute example.

```
1  /* J is an output kernel parameter  */
2  int J[1000];
3  /* Indices 100 to 150 from K are *
4   * output for the kernel         */
5  int K[1000];
6  /* L is a multiarray output kernel parameter */
7  int L[100][100];
8
9  [[ rpr::kernel,
10    rpr::out(J[1000],K[100:100],L[100][100]),
11          ... /* rest of attributes */ ]]
12 for(i=0; i<1000; ++i) {
13     ...
14     J[i] = i++;
15     K[100 + (i%2)] = i++;
16         L[i][i] = 1;
17     ...
18 }
```

execution (in sequential code). The data could be kept data until the next kernel execution that uses this variable.

Figure 4.1 shows a sample of the lifetime of a variable during the execution of several kernels.

This attribute can be used in combination with *rpr::in* and *rpr::out* parameter attributes, defining variables that are only used by kernels:

- If a *rpr::keep* variable is defined also as *rpr::in*, and it is not in device memory, the variable will be transferred from host to the device, and it may be kept at the end of kernel execution if the next kernel is executed on the same device. Otherwise, it may be released.
- If a *rpr::keep* variable is defined also as *rpr::out*, the variable may be created previously in target device memory. If the next kernel is not executed in the same device, at the end of the kernel execution the variable will be transferred from the device to the host, and then deleted from device memory.

A keep variable will be deleted from a device when this variable is not included as keep in next kernels (See Figure 4.2).

All the *rpr::keep* variables stored in devices will be released in the last defined kernel of a program.

The syntax of *rpr::keep* attribute is as follows:

[[ **rpr::kernel, rpr::keep(** `variable` , *var-in-list* ) , *rpr-attribute-list* ]]

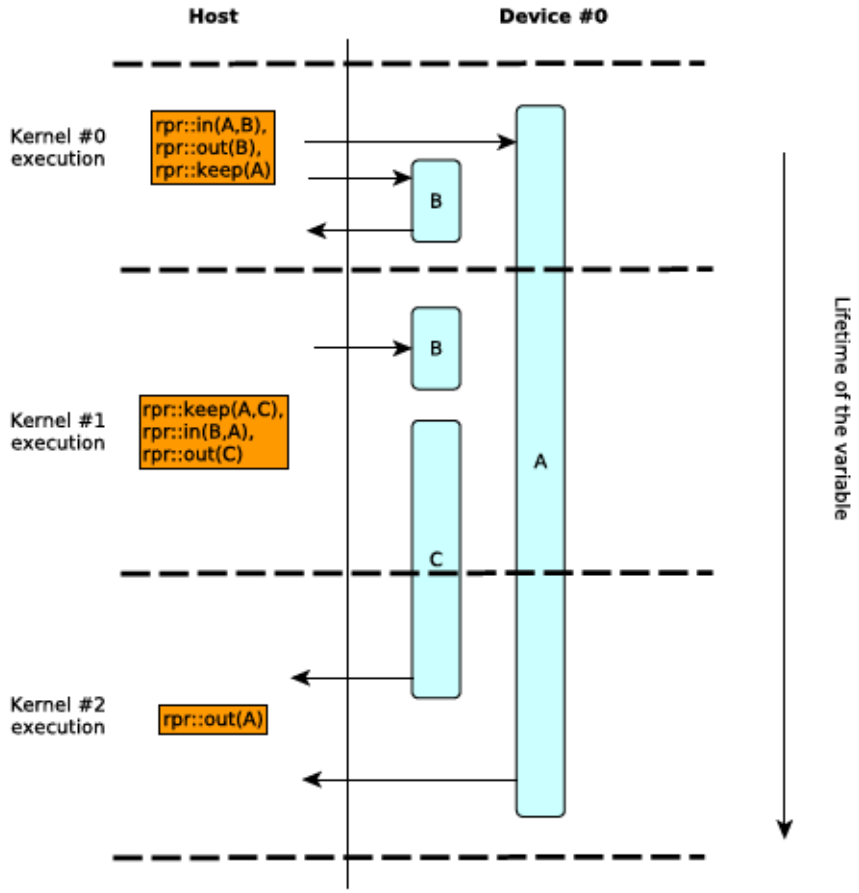


Figure 4.1: Lifetime of a variable using `rpr::keep` attribute.

```
kernel region
```

The arguments of the attribute are the list of variables, separated with commas, that act as kept variables in the device memory at the end of kernel execution.

Regarding the use of the *rpr::keep* attribute:

- can only be included once per kernel attribute definition.
- can only appear in the definition of a kernel.
- is optional. If *rpr::keep* contradicts execution plan defined by the scheduler, the attribute is ignored.

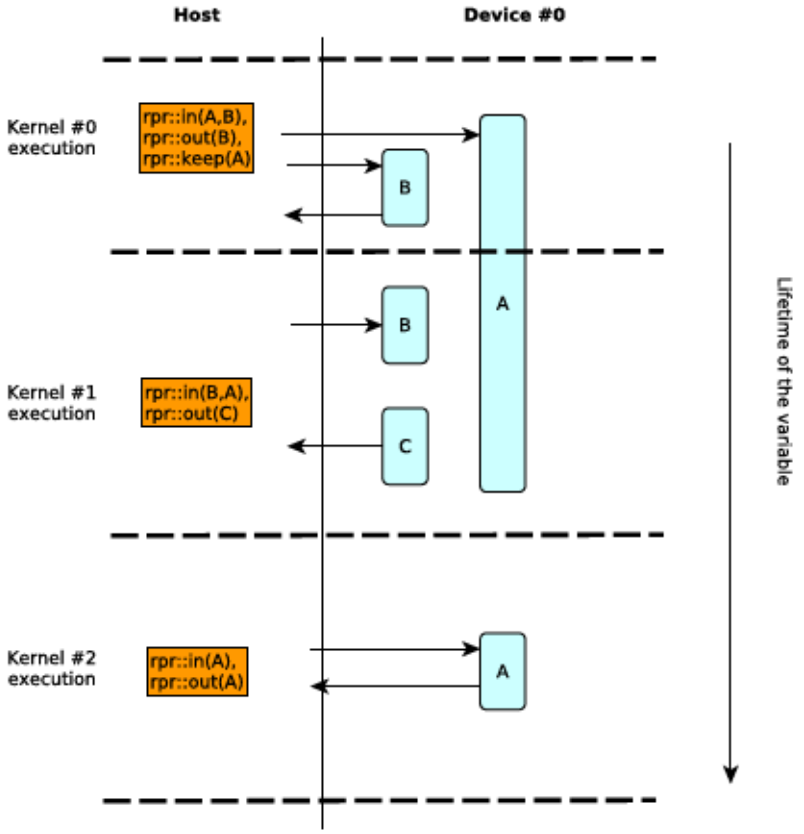


Figure 4.2: Lifetime of some variables using `rpr::keep` attribute.

- the scheduler will not consider misuse of *rpr::keep* attribute. For example, if the user suggests that a variable be kept in a device between kernels, but it is modified in the host between them, the behaviour will not be controlled.
- is ignored in the last defined kernel, because after the last execution all the device memory reserved must be released.

Listing 4.10 is an example of *keep* attribute where the `A` kernel variable may be kept in the computing device across multiple kernel execution.

Listing 4.11 is an example of *keep* attribute where `A` and `B` variable may be kept in the computing device across multiple kernel execution. The first kernels keep data until the last kernel if it is possible.

Listing 4.12 is an example of *keep* attribute where there is an unexpected behaviour because a variable stored in a device could change its value in host memory.

Listing 4.10: keep attribute example where a variable is kept in the previous computing device.

```
1  int A[N][M];
2  /* rest of the code
                              */
3
4  /* Matrix A is transfered to the device and, if not in  *
5   * conflict with the scheduler, kept at the end of                *
6   * kernel execution
                 */
7  [[ rpr::kernel, rpr::in(A), rpr::keep(A),
8           ... /* rest of attributes */ ]]
9  for(i=0; i<N; ++i) {
10          ...
11 }
12 ...
13 /* part of the host execution where "A" is not used      */
14 ...
15
16 /* "A" variable is not transfered to the kernel. Again  *
17  * the scheduler tries to keep in the device memory      */
18 [[ rpr::kernel, rpr::in(A), rpr::out(A), rpr::keep(A),
19          ... /* rest of attributes */ ]]
20 for(i=0; i<M; ++i) {
21          ...
22 }
23 ...
24 /* "A" if variable resides in device memory, it will    *
25  * not be transfered from host
                    *
26  * "A" variable will be transfered to host and deleted  *
27  * at the end of kernel execution.
                       */
28 [[ rpr::kernel, rpr::out(A),
29          ... /* rest of attributes */ ]]
30 for(i=0; i<M; ++i) {
31          ...
32 }
```

### rpr::size

*rpr::size* defines the problem size of the kernel. For example, this attribute may specify the number of iterations of the associated loops of a kernel. In case of multiple loops it is possible to define multiple values split by commas, one per each dimension of the problem.

The syntax of *rpr::size* attribute is as follows:

Listing 4.11: keep attribute example where a variable is kept in the previous computing device.

```
1  int A[N][M];
2  int B[M];
3
4  /* Matrix "A" is an output of kernel. It will keep, if  *
5   * it is possible, until next kernel that use it as     *
6   * parameter (thrid kernel in this case).               */
7   [[ rpr::kernel, rpr::out(A), rpr::keep(A),
8           ... /* rest of attributes */ ]]
9  for(i=0; i<N; ++i) {
10          ...
11 }
12 ...
13 /* "B" is used by the kernel and kept if it is possible */
14 [[ rpr::kernel, rpr::in(B), rpr::out(B), rpr::keep(B),
15          ... /* rest of attributes */ ]]
16 for(i=0; i<M; ++i) {
17          ...
18 }
19 ...
20 /* "A" and "B" variables may reside in device memory.   */
21 [[ rpr::kernel, rpr::in(A,B),
22          ... /* rest of attributes */ ]]
23 for(i=0; i<M; ++i) {
24          ...
25 }
```

[[ **rpr::kernel, rpr::size(** `size` , *size-list* **)** , *rpr-attribute-list* ]]
*kernel region*

`size` represents the number of elements to be processed. *size-list* represents a list of sizes used by different dimensions of the problem.

The *rpr::size* attribute:

- is mandatory for kernel region definition.

Listing 4.13 is a basic example of *rpr::size* attribute.

Listing 4.14 shows an example where *rpr::size* attribute for 2-dimensional problem.

### rpr::pattern

The *pattern* clause is used to specify the pattern in which memory is accessed in a kernel.

Listing 4.12: Wrong sample of `keep` attribute where a variable is modified.

```
int A[N][M];
/* A variable is used outside of kernel          *
 * execution but kept in device memory           */
[[ rpr::kernel, rpr::in(A), rpr::keep(A),
        ... /* rest of attributes */ ]]
for(i=0; i<M; ++i) {
        ...
}
...
A[i][j] = value;
...

/* ERROR: "A" device variable and "A" host   *
 * variable are different                                    */
[[ rpr::kernel, rpr::in(A),
        ... /* rest of attributes */ ]]
for(i=0; i<M; ++i) {
        ...
}
```

Listing 4.13: `size` attribute example.

```

/* A kernel with 1000 elements *
 * to be processed              */
 [ rpr::kernel,
   rpr::size(1000),
   ...
]]
for(int i=0; i<N; ++i) {
...
}
```

Figure 4.3 shows different patterns using 2D representation.
The syntax of *rpr::pattern* attribute is as follows:

[[ **rpr::kernel, rpr::pattern**(`var-pattern`, *var-pattern-list*)
, *rpr-attribute-list* ]]
*kernel-region*

Where `var-pattern` is composed by:

- the name of the array or vector variable affected,

Listing 4.14: `size` attribute example for 2d problem.

```
/* Execution of 2d kernel from  *
 * 0 to 1000 and from 0 to 100  *
 * in each dimension            */
[       rpr::kernel,
        rpr::size(1000, 100),
        ...
]]
for(auto i=0; i<1000; ++i) {
        for(auto j=0; j<100; ++j) {
        ...
        }
}
```

- the *pattern* defined using `Intel array notation` [66]. This notation represents neighbours of an array element taking the lower-bound the relative position of the selected element.

For complex, patterns are possible to join individual patterns in user-defined list. The *rpr::pattern* attribute:

- can only be included once per kernel attribute definition.
- can only appear in the definition of a kernel.
- can be only used with arrays and vectors.

Listing 4.15 explains two different uses of the *pattern* clause.

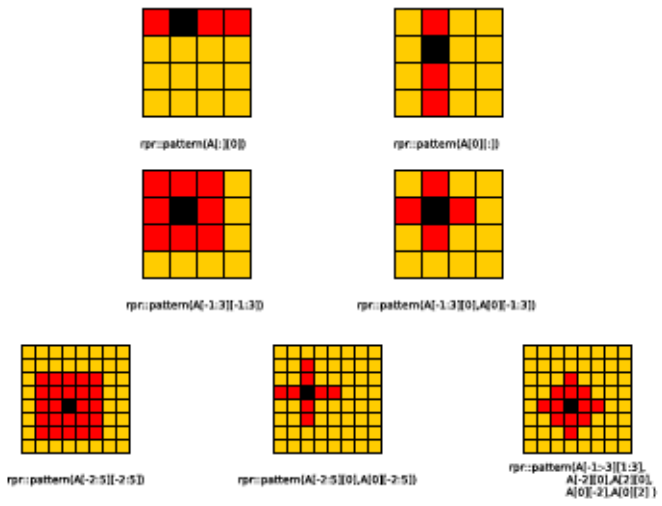


Figure 4.3: Samples of patterns using *rpr::pattern* attribute.

Listing 4.15: `pattern` attribute example.

```
/* The kernel will access a matrix in a cross patterns, *
 * from top to bottom and from left to right            */
[[ rpr::kernel,rpr::pattern(A[:][0],A[0][:]),  ...]]
for(i=0; i<N; ++i) {
    for(j=0; j<M; j++) {
        ....
        }
}

/* The kernel will add all the neighbors of an element  *
 * with a window of radius 3                            */
[[ rpr::kernel,rpr::pattern(A2d[-3:7][-3:7]), ...]]
for(i=0; i<N; ++i) {
    for(j=0; j<M; j++) {
                A2d[i][j] = A2d[i][j]+A2d[i+1][j]+A2d[i+2][j]
                +A2d[i+3][j]+A2d[i][j+1]+A2d[i+1][j+1]
                +A2d[i+2][j+1]+A2d[i+3][j+1]...;
        }
}

/* The kernel will use the elements place in a plane    *
 *  of a 3d matrix                                      */
[[ rpr::kernel,rpr::pattern(A3d[:][:][0]), ...]]
for(i=0; i<D1; ++i) {
    for(j=0; j<D2; j++) {
        for(k=0; k<D3; k++) {
                        ...
                }
        }
}
```

### rpr::reduce

It specifies a reduction operator and one or more scalar variables. For each variable, a private copy is created for each parallel thread of the defined and initialized for that operator. At the end of the region, the values for each thread are combined using the reduction operator, and the result combined with the value of the original variable and stored in the original variable. The reduction result is available after the kernel execution.

Table 4.2 shows the operators that are valid and the initialization values; in each case, the initialization value will be cast into the variable type. The name of these operators is built using the *rpr* namespace. For *max* and *min* reductions, the initialization values are the least representable value and the largest representable value for the data type of the variables, respectively. Supported data types are the numerical data types in C++ (e.g. int, float, double, complex).

Table 4.2: Operators used with *reduce* attribute.

| operator | Initial value | Description |
|---|---|---|
| add | 0 | Addition |
| mult | 0 | Multiplication |
| max | smallest represented value | Maximum value |
| min | largest represented value | Minimun value |
| band | $\tilde{0}$ | Binary AND operation |
| bor | 0 | Binary OR operation |
| bxor | 0 | Binary XOR operation |
| and | 1 | Logical AND operation |
| or | 0 | Logical OR operation |

**Syntax** The syntax of *rpr::reduce* attribute is shown as follows:

```
   [[ rpr::kernel, rpr::reduce(operation, variable )
, rpr-attribute-list]]
        kernel region
```

Regarding the use of the *rpr::reduce* attribute:

- it can only be included once per kernel attribute definition.
- it must be included in a kernel region definition.
- it can only appear in the definition of a kernel.
- there is only one *reduce* attribute per *kernel* definition.
- *reduce* defined variables are considered as *rpr::out* variables in terms of kernel parameters.

Listing 4.16 is an example of *reduce* attribute.

### 4.1.4 High-level parallel patterns

Parallel patterns typically have a general implementation that works well, and is included in many parallel frameworks (e.g. pipelines, maps). Therefore, it is a

Listing 4.16: `reduce` attribute example.

```
1
2 int var=0;
3 /* Example on the use of the reduce attribute */
4 [[ rpr::kernel,rpr::reduce (add, var),
5    ... /* rest of attributes */ ]]
6 for(i=0; i<N; ++i) {
7   ...
8   var+=value[i];
9   ...
10 }
```

desirable feature that users can identify such parallel patterns in their code. Our proposed annotations provide a simple syntax with which users can introduce high-level parallel patterns in the source code.

### rpr::pipeline

*rpr::pipeline* attribute defined a pipeline skeleton. A pipeline is composed of several stages that have to be executed in a particular order. Data flows through a series of pipeline stages and each stage processes the data in some way. These stages are marked as kernels and the data are defined in *rpr::stream* attribute.

The stream programming model decomposes programs into tasks/kernels and indicating the data flow among them. This exposes data, task and pipeline parallelism. A pipeline is composed of:

- Stages of the pipeline, defined by *rpr::kernel* attribute.
- Data stream defines by *rpr::stream* attribute. In combination with *rpr::in* and *rpr::out* attributes, it provides a way to expose producer-consumer relationships in a pipeline.

Usually, the data stream is created at the beginning of the first stage of the pipeline. Produced data will be defined as output parameter in the first kernel of the pipeline. The data stream will be deleted at the end of the last stage of the pipeline, including total or partial data stream.

The pipeline uses the loop condition to determine the end of the stream. In case of nested loops, the pipeline will use the combination of the loop conditions defined below pipeline definition.

The semantic of streaming input and output attributes ensures that inputs are buffered until the next activation of the producer kernel, and outputs are buffered until the next activation of consumer kernel.

Furthermore, it is possible to specify in a kernel the order in which parameters are received and results are sent to and from a stage made of multiple workers defined

with *rpr::farm* attribute (see below). In these cases, it is necessary to make clear if the data have to be received or sent in order, or if, on other hand, it can be done out of order. Whenever the data produced/consumed must follow a sequential order, then the pipeline is *in-order*. Alternatively, a stage of the pipeline is *out-of-order* if there is no strict order for the parameters/results to follow when consumed or produced, respectively. *out-of-order* is the behaviour by default.

Figure 4.4 shows a sample of `rpr::pipeline` attribute.

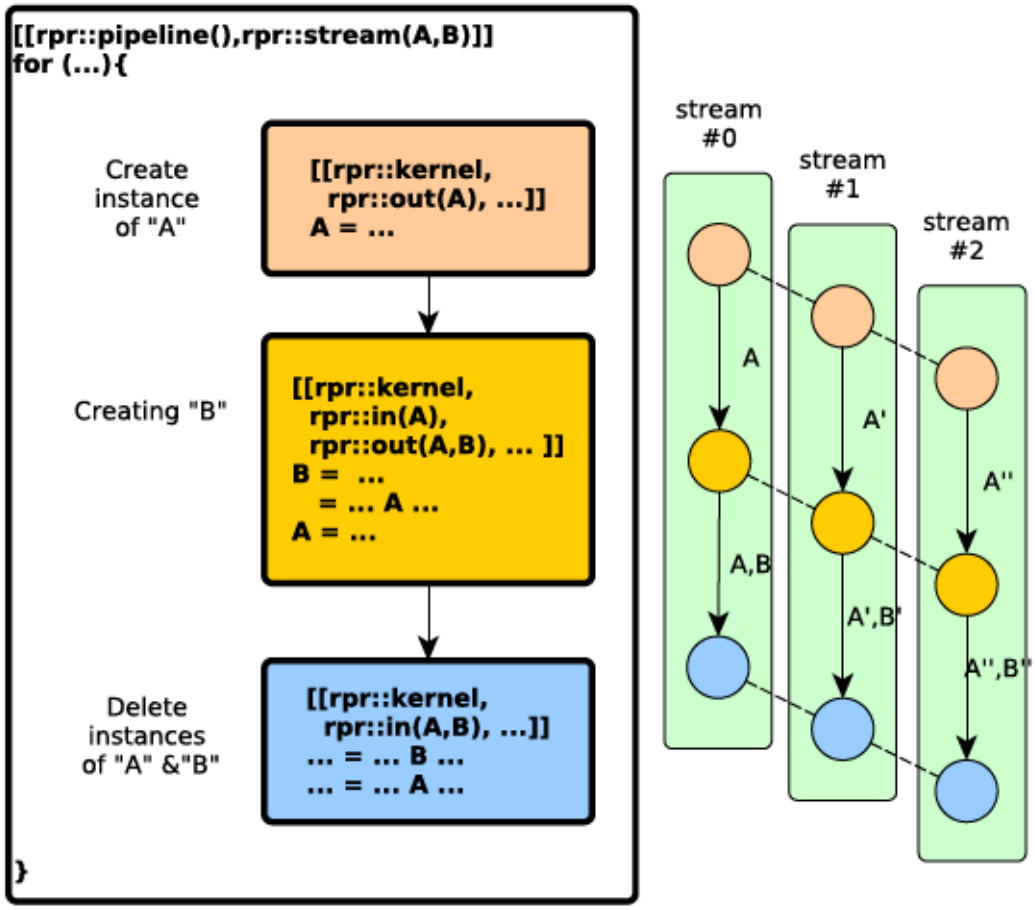


Figure 4.4: Sample of *rpr::pipeline* attribute.

Figure 4.5 shows a sample of `rpr::pipeline` attribute working in combination with `rpr::farm`.

**Syntax** The syntax of *rpr::pipeline* attribute is as follows:

[[**rpr::pipeline(** *type, size, use* **), rpr::stream(** `variable` , *var-list* **)** , *rpr-*

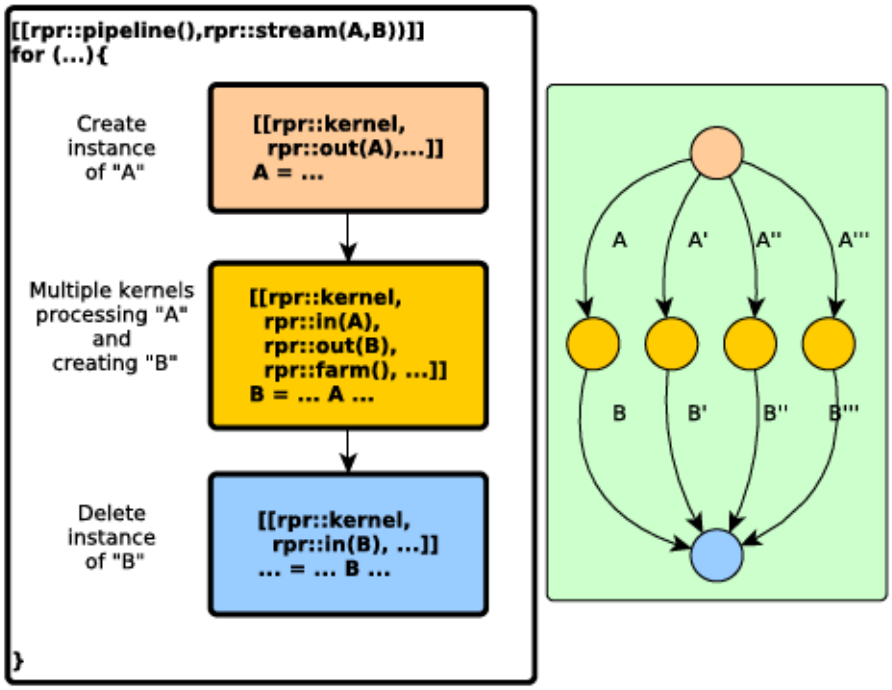


Figure 4.5: Sample of *rpr::pipeline* attribute with *rpr::farm*.

*attribute-list*]]

```
loop/nested loop statement
    kernel regions
```

where *type*, *size* and *use* are optional parameters of *rpr::pipeline*, meanwhile `variable` and *var-list* defined in *rpr::stream* are the variables that represents the stream in the pipeline.

*type* is optional and it could be `bound` or `unbound`. It means the behaviour of the buffers used between the stages of the pipeline. By default, the value of `type` parameter is `unbound`.

`size` is mandatory if *type* is `bound`. It represents the maximum size of elements of the internal buffers used in the pipeline placed between kernels.

Finally, `use` parameter is optional and it may appear if `size` is defined before. `use` parameter takes these values: `blocking` or `non-blocking`. It represents the accessing behaviour when data are included in the internal buffers placed between kernels. If it is not included, the default value is `non-blocking`.

Regarding the use of the *rpr::pipeline* and *rpr::stream* attributes:

- they can only be included once per pipeline attribute definition.

- *rpr::pipeline* can only be used with loop statements.
- all the statements defined in a pipeline must be part of a kernel. Otherwise, the pipeline construction is illegal.
- the pipeline must have only one data flow. *If-else* statements are not allowed inside a pipeline.
- the stream parameters must be part of:
  - output parameters of the first kernel.
  - input/output parameters of the kernels, creating a work-flow.
  - input parameters of the last kernel.
- if the stream parameters are part of output parameters of the last kernel, the last element generated by the pipeline could be accessed outside of the pipeline execution.

Listing 4.17 is a basic example of pipeline. Listing 4.18 is an example containing a pipeline working in combination with farm skeleton. Listing 4.19 is an example containing a wrong definition of a pipeline.

#### rpr::farm

This attribute represents *task parallelism*. Farm represents the execution of different tasks by the same farm that are replicated and executed in parallel. As the calculations are independent, no information needs to be exchanged between tasks during this time, and sharing of the results can be postponed until all the tasks have completed.

The behaviour of the output is defined using the *out-behaviour* parameter of the attribute. If the parameter is not included in the attribute, the output is defined as `unordered` by default.

The syntax of *rpr::farm* attribute is as shown as follows:

[[ **rpr::kernel, rpr::farm**(*replicas, out-behaviour*), *rpr-attribute-list*]]
`kernel region`

where `replicas` is a number that indicates the number of workers used in the farm and `out-behaviour` defines the order of the output of the farm. By default, `replicas` takes the maximum number of workers in the platform. `out-behaviour` could take these values: `ordered` and `unordered`, that is taken by default.

The *rpr::farm* attribute:

Listing 4.17: `pipeline` attribute example.

```
object A,B;
/* loop statement */
[[rpr::pipeline (), rpr::stream (A, B)]]
for (...){
        /* Information used by the stream will be       *
         * created here
                                */

        /* first stage of pipeline */
        [[rpr::kernel, rpr::out(A),
      ... /* rest of attributes */ ]]{
                A = ...;
        }
        /* second stage of pipeline */
        [[rpr::kernel, rpr::in (A), rpr::out (B),
      ... /* rest of attributes */ ]]{
                B = ...A...;
                ...

        }
        /* third stage of pipeline */
        [[rpr::kernel, rpr::in (B),
      ... /* rest of attributes */ ]]{
                f(B);
                ...
        }
        /* Information used by the stream will be deleted here */
}
```

- can only be included once per kernel attribute definition.
- can only appear in the definition of a kernel.
- is incompatible with `rpr::map`.

Listing 4.20 is an example of *rpr::farm* attribute.

### rpr::map

**Summary** The `rpr::map` attribute indicates split, execute, merge computation. Map skeleton represents *data parallelism* execution. It is the simultaneous execution on multiple instances of the same function across the elements of a dataset.

The syntax of *rpr::map* attribute is as shown as follows:

[[ **rpr::kernel, rpr::map**(*replicas*), *rpr-attribute-list*]]

Listing 4.18: `pipeline` attribute example with farm.

```
object A, B, C;
/* loop statement                                  *
 *  A and B are part of the stream                 *
 *  The internal buffers of the pipeline are:      *
 *       - Bound internal buffers                      *
 *       - Number of elements stored of internal       *
 *       buffers: 8                                    *
 *       - Blocked access of internal buffers          */

[[rpr::pipeline (bound, 8, blocking),
  rpr::stream (A, B)]]
for (...){
        /* Stage of the pipeline */
        [[rpr::kernel, rpr::out(A), ...]] {
                A = f1();
        }
        /* Stage of the pipeline
                    *
         * 4 workers and generates output in-order      */
        [[rpr::kernel,
      rpr::farm(4, ordered),
      rpr::in(A,C),
      rpr::out(A,B),... ]]
        for(i=0; i<M; ++i) {
                ...
                B = ...A...;
                ...
        }

        /* Stage of the pipeline */
        [[rpr::kernel,
      rpr::in(A,B),...]] {
                f3(A,B);
                ...
        }
}
```

`kernel region`

where `replicas` is a number that indicates the number of workers used in the map. By default, `replicas` takes the maximum number of workers in the platform.

The *rpr::map* attribute:

- can only be included once per kernel attribute definition.
- can only appear in the definition of a kernel.

Listing 4.19: `pipeline` attribute wrong sample.

```
object A,B,C;

/* loop statement */
[[rpr::pipeline (), rpr::stream (A, B)]]
while(true){
        /* Information used by the stream will  *
         * be create here
                    *
         * First stage of pipeline                                    */
        [[rpr::kernel, rpr::out(A), ...]] {
                A = ...;
        }
        /* Second stage of pipeline                          */
        [[rpr::kernel, rpr::in (A), rpr::out (B), ... ]] {
                B = ...A...;
                ...

        }
        /* Third stage of pipeline                                    */
        // Illegal stream variables are not
        // included in kernel parameters
        [[rpr::kernel, rpr::in (C), ... ]] {
                f(C);
                ...
        }
        // Illegal
        if (condition)
                i++;
}
```

- is incompatible with `rpr::farm`.

Listing 4.21 is an example of the *rpr::map* attribute.

### 4.1.5 Utility attributes

The utility attributes are only used as a means to ask for debug information. For now, only a logging attribute is defined.

#### rpr::log

**Summary** *log* attribute shows profiling information about a marked kernel with this attribute. The information to show is configurable by the developer as a list of profiling parameters. This profiling information will be stored in an output file json file named as: `<nameofapplication>.json`

Listing 4.20: `farm` attribute example.

```
/* Farm by default using                                    *
 * maximum number of workers and          *
 * unordered output                                                  */
[[rpr::kernel,
  rpr::farm(),
   ... /* rest of attributes */ ]]
for(i=0; i<N; ++i) {
   for(j=0; j<N; ++j) {
        ...
   }
}
/* Farm with 4 workers and output       *
 * ordered
                */
[[rpr::kernel,
  rpr::farm(4, ordered),
   ... /* rest of attributes */ ]]
for(i=0; i<N; ++i) {
   for(j=0; j<N; ++j) {
        ...
   }
}
```

Listing 4.21: `farm` attribute example.

```
/* Map kernel using maximum number      *
 * of workers by default.                        */
[[rpr::kernel,  rpr::map(),
   ... /* rest of attributes */ ]]
for(i=0; i<N; ++i) {
   for(j=0; j<N; ++j) {
        ...
   }
}
/* Map with 4 workers                            */
[[rpr::kernel,  rpr::map(4),
   ... /* rest of attributes */ ]]
for(i=0; i<N; ++i) {
   for(j=0; j<N; ++j) {
        ...
   }
}
```

Listing 4.22 represents the schema of the output *log* file.

Listing 4.23 shows the schema of the *inparams* and *outparams* parameters in log file.

Listing 4.22: `log` output file format.

```
{
        "applicationid":<uint>,
        "kernelid":<uint>,
        "file":<string>,
        "line":<uint>,
        "log":[
                {
                        "class":<string>,
                        //'class' may take one of these values:
                        //- inparam
                        //- outparam
                        //- keepvar
                        //- ktime
                        //- etime
                        //- stime
                        //- rtime
                        //- sendvars
                        //- recvvars
                        //- device
                },
                {...}
        ]
}
```

The syntax of *rpr::log* attribute is as shown as follows:

[[ **rpr::kernel, rpr::log(*log-parameter-list*)**, *rpr-attribute-list* ]]
`kernel region`

where *parameter* could take one or more of the following values depending the information wanted from a kernel:

- `inparams`: Input parameters of a kernel, showing for each variable:
  - variable identification
  - name
  - offset and size for each dimension
  - total size of the variable in bytes
  - type (i.e. float, double, int)
- `outparams`: Output parameters of a kernel, showing for each variable:
  - variable identification
  - name
  - offset and size for each dimension

Listing 4.23: `inparams` and `outparams` parameters representation in log file.

```
{
        "class":"inparams",
        "variable":[
                //one per variable
                //it could be empty.
                {
                        "id":<uint>,
                        "name":<string>,
                        "totalsize":<uint>,
                        "type":<string>,
                        "dimension":[
                                //one per each dimension.
                                //it could be zero
                                {
                                        "dim":<uint>,
                                        "size":<uint>,
                                        "offset":<uint>
                                },
                                {....}
                        ]
                },
                {....}
        ]
},
{
        "class":"outparams",
        "variable":[
                //one per variable
                //it could be empty.
                {
                        "id":<uint>,
                        "name":<string>,
                        "totalsize":<uint>,
                        "type":<string>,
                        "dimension":[
                                //one per each dimension.
                                //it could be zero
                                {
                                        "dim":<uint>,
                                        "size":<uint>,
                                        "offset":<uint>
                                },
                                {....}
                        ]
                },
                {....}
        ]
}
```

  - total size of the variable in bytes
  - type (i.e. float, double, int)

- `keepvars`: Kept data after kernel execution. The information presented for each kept variable is:
  - variable identification

  - name
  - size in bytes
  - type

- `ktime`: Kernel execution time in seconds.
- `etime`: Execution time in seconds. It includes send, kernel execution and recibe time operations.
- `ttime`: Transfer time in seconds. It includes send and recibe time operations.
- `stime`: Send transfer operation time in seconds.
- `rtime`: Receive transfer operation time in seconds.
- `sendvars`: It shows the information about the data sent to devices in a kernel. The information to show for each variable is:
  - variable identification
  - name
  - offset and size for each dimension
  - total size of the variable in bytes
  - type (i.e. float, double, int)
  - device
- `recvvars`: It shows the information about the data received from devices in a kernel. The information to show for each variable is:
  - variable identification
  - name
  - offset and size for each dimension
  - total size of the variable in bytes
  - type (i.e. float, double, int)
  - device
- `range`: It shows number of elements processed in a device.

Regarding the use of the *rpr::log* attribute:

- it can only be included once per kernel attribute definition.
- it can only appear in the definition of a kernel (See Section 4.1.2).
- its profiling only covers the kernel.

Listing 4.24 is an example of the usage of debug attribute.

Listing 4.24: `log` attribute example.

```
1  /* Examples on the use of the debug attribute.
                                *
2   * It will show the parameters and their type (i.e. in, out, inout)
       , *
3   * characteristics (e.g. type of variable, size).
                              *
4   * Also, It will show the target device where the code runs and the
         *
5   * execution time at the end of the kernel execution
                            */
6  [[ rpr::kernel,
7     rpr:log(inparams, outparams, etime),
8     ... /* rest of attributes */ ]]
9  for(i=0; i<N; ++i) {
10           ...
11 }
```

## 4.2 Automatic annotation techniques

Manually annotating the code can be an expensive and usually error-prone process, depending on the complexity of the code. Developers need to identify hotspots in the original source code. However, not every detected hotspot can be offloaded due to internal data dependencies. Hotspots that can be parallelized are annotated as `kernels`. To obtain the best possible performance, developers need to understand the characteristics of devices that make up the heterogeneous parallel platform. To determine if a kernel could be offloaded in a specific heterogeneous device, some restrictions have to be taken into account, such as memory allocations or system calls. Finally, there is another important consideration: data transfer operations between devices. These transactions consume time resources that should be avoided as much as possible in order to improve the performance. Furthermore, data dependency analysis through kernels is needed for minimizing data transfers.

In light of this, as a complement to the annotation specification, we propose an set of automatic annotation techniques, called the Automatic Kernel Identification tool (AKI).

AKI is a semi-automated process that aims to detect potential source code regions, which can be ported to accelerators-based kernels in applications developed in C++. Once potential kernels are detected, their characteristics are studied and the possible devices on which they can be run are identified. At this point, developers can implement these regions on specific oriented accelerator languages such as CUDA, OpenCL, and Intel LEO [67]. The main goal of AKI is to reduce the programming effort by reducing the necessity of manual source code re-implementation for parallel heterogeneous platforms. For now, the tool can automatically detect

and annotate kernels, but any extra information (e.g. data dependencies, high-level parallel patterns) must be introduced manually. This process can be made fully automated, but for the scope of this Thesis, the tool only annotates kernels. AKI relies on scripting (shell scripting and AWK, mainly), as well as some external tools, in order to analyse and annotate the original source code.

### 4.2.1 Workflow

The proposed attributes are added to the source code by a sequence of stages. The input of the AKI workflow is the original source code, and the output is the annotated source code. Figure 4.6 details the five workflow stages:

1. **Hotspot detection:** The source code is inspected automatically in order to detect hotspots.
2. **Hotspot selection:** Hotspots are filtered in regards to the rules established by the domain expert.
3. **Preliminary kernel annotation:** The hotspots detected in the previous step are tagged for the following stages. Kernels are annotated using the `rpr::kernel` attribute.
4. **Kernel selection:** Kernels are analysed in order to select the most promising ones. For example, in this step, the innermost loops in a nested loop are discarded.
5. **Attribute annotation:** The rest of the proposed C++ attributes are included in the selected kernels obtained from the previous stage. For example, it is in this stage that incompatible devices are discarded for each kernel.

AKI allows improving the result of some stages using the developer knowledge, making AKI a semi-automatic tool. The developer can customize the following stages: hotspot selection, kernel selection, and automatic attribute annotation. The following subsections explain in details each of AKI workflow stages.

### 4.2.2 Hotspot detection

Hotspots regions are code blocks where the application spends more computational resources. By optimizing these application regions, we can improve the overall execution time of the applications.

Hotspots cannot be determined using a source code analysis (aka static analysis), given that of the number of loop iterations could be unknown at compilation time.

AKI relies on a runtime analysis approach (aka dynamic analysis) for detecting hotspots. The code instrumentation uses the following base tools: *gcov* [80],

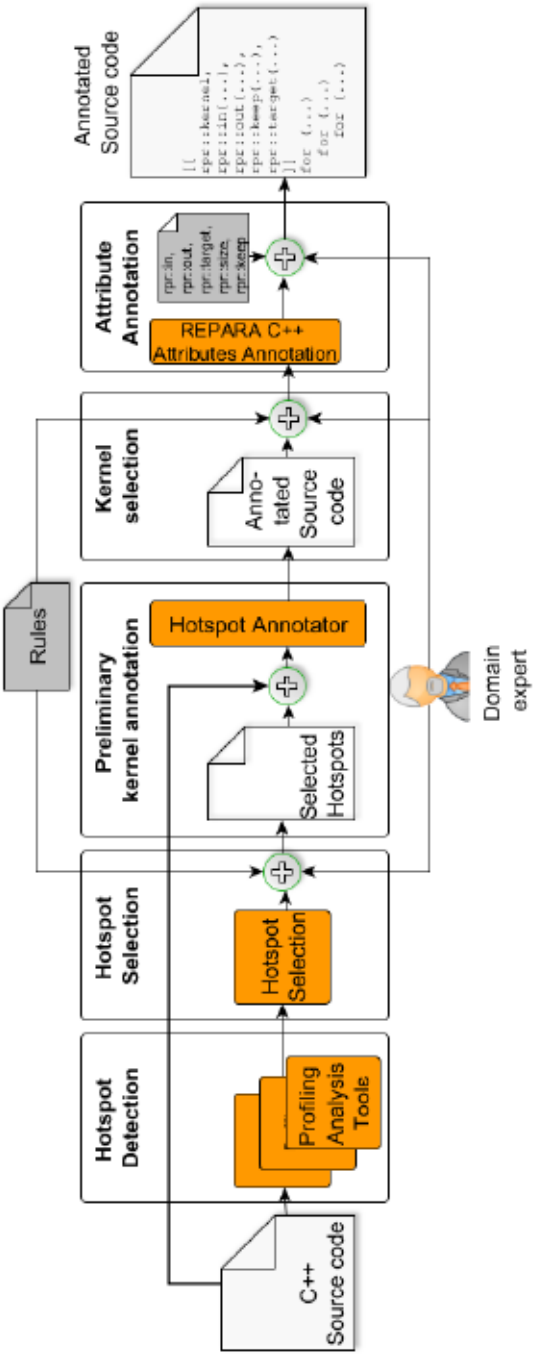


Figure 4.6: Overview of Automatic Kernel Identification (AKI) workflow.

*gprof* [53], and *oprofile* [78]; *gcov* tool provides the number of invocations for each source code line. *gprof* tool provides the time spend per function. The *oprofile* tool leverages the hardware performance counters.

The generated execution profiling data is automatically aggregated and analysed in order to detect code region candidates to be annotated as kernels. The output of each tool is processed and transformed into processable data. All data is combined in a table with the information needed for further AKI stages.

In C++, generic programming based on templates is commonly used (e.g. STL algorithms/containers). When a template is instantiated, the execution of the resulting code might be a hotspot. In this case, AKI selects the template source code as potential hotspot when at least one instantiation of this template is detected as such by the AKI base tools used in previous steps.

The output information provided by the aforementioned tools is automatically combined and ordered in order to identify the code hotspots. These are used for the next stage of our proposed workflow: the hotspot selection.

### 4.2.3 Hotspot selection

This stage applies a selection process to find the candidate regions for being optimized. The selection process is described on Algorithm 1. The initial work described in this document targets loop-based hotspots.

**Algorithm 1** Block code selection process.

```
1: function BlockDetect(annotated_source, threshold)  ▷ annotated_source is
   a vector with the annotated source code
2:     max[time] = max[executions] = 0;
3:     for each line ∈ annotated_source do
4:         if line[executions] > max[executions] then
5:             max[executions] = line[executions];
6:         if line[time] > max[time] then
7:             max[time] = line[time]
8:     for each line ∈ annotated_source do
9:         line[weight] = (line[executions]/max[executions])*we +
10:                        (line[time]/max[time])*wt;
11:        if isA(line[source],loop) & line[weight] > threshold; then
12:            annotate(line,rpr::kernel);
```

The algorithm has two inputs: the obtained code regions from the former stage and the threshold provided by the domain expert. This threshold represents the sensibility ratio of the algorithm for tagging a hotspot as a kernel. $we$ and $wt$ represent the weights used by executions and time respectively, where $we + wt = 1$ and $0 \leq we, wt \leq 1$. The algorithm produces as a result an annotated version of the source code with a list of potential kernels, defined by the first and last line of each region.

### 4.2.4 Preliminary kernel annotation

This stage consists in annotating the detected hotspots as kernels. At this stage, AKI annotates these kernels by using the `rpr::kernel` attribute. This attribute indicates that the following loop can be used as a kernel or task in a similar way to CUDA, OpenCL or OpenMP. The idea is to find all those code regions that have passed the hotspot selection so they can be annotated as `rpr::kernel`. The attribute is placed before the line of code where the selected code region starts. In the case of nested-loops, we consider a coarse-grain approach. Domain experts can customize the preliminary kernel annotation by modifying the grain size, for example from coarse-grain to fine-grain.

### 4.2.5 Kernel selection

As described before, the goal of this stage is to discard the least promising kernels. AKI filters the detected kernels in order to remove them, following an established criteria: code complexity and data dependencies with other preliminary kernels. This filter is based on static code analysis of the promising kernels.

Our first approach will be a naive filtering, where AKI distinguishes between those loops that initialize arrays and those with an greater computational cost. For this purpose, AKI relies on Columbus [48], which obtains static source code metrics. The metrics used to analyse the complexity are the following:

- **LLOC**: Number of logical source code lines. We assume that an initialization loop will have a small computational overhead, as opposed to a traditional for loop that does some processing over some variable, such an array.
- **NLE**: Nesting level else-if. We assume that an initialization loop contains a low value of this metric.
- **EXP**: Number of expressions in a kernel. We define a threshold to discard small computational regions in order to avoid offloading overheads.
- **MUL/EXP** and **ADD/EXP**: Ratio of multiplications and additions based on the **EXP** value of a kernel. We define another threshold that filters kernels with small computational load.

Metrics related to memory transfers (e.g. memory bandwidth, memory access patterns) are not considered in this work. The result of this stage is the initial source code with the final set of annotated kernels. These kernels can later studied in the annotation phase in order to include additional attributes such as target devices and in-out variables. The result of this process will be the final annotated code.

### 4.2.6 Attribute annotation

The result of the previous phase is a preliminary version of the annotated code, containing the annotations for the kernel attributes. In this phase, more information can be added to the annotations, such as preferred devices, data dependencies or parallel patterns. For now, given the programming cost, this is a manual process, although it can be automated by using a more in depth analysis of the code. For example, the Clang/LLVM compiler infrastructure provides functions that can simplify the analysis.

## 4.3 Summary

In this chapter we have proposed a software annotation syntax, as well as a set of automatic annotation techniques that can be used to automatically detect and annotate kernels. Together with the contributions proposed in the previous chapter, we have all the necessary inputs for the task partitioning techniques. In the following chapter, we will show how the task partitioning techniques work, as well as the role that the architecture description and the software annotations play in the process.

# Chapter 5

# Static partitioning techniques

So far, we have discussed the underlying model and the kernel identification and annotation techniques proposed in this Thesis. Both are used to produce the necessary information for the partitioning techniques. In this Chapter, we present a partitioning algorithm that makes use of the work developed so far.

## 5.1 Task partitioning algorithm

So far, we have defined a hardware description language, a software annotation specification, along with automatic annotation techniques, and a model that acts as a base for all of these. The last part of the whole are the partitioning techniques. These techniques will take the information about the hardware architecture, the software annotations, and the static metrics of the code, and will schedule the kernels in the available devices according to the execution model.

In this section we propose an approach towards software partitioning. This approach will work statically and will serve as the basis for the more advanced partitioning techniques. It confirms the feasibility of the proposed static partitioning algorithm.

The objective of this algorithm is to schedule the set of kernels from a codebase into the computing devices in an heterogeneous platform. To this end, a model has been created for representing all the valid schedules. The model is based on four key elements:

- **Devices**. Compilation available devices and their characteristics as extracted from the hardware description.
- **Kernels**. This is the list of kernels to be run as extracted from the software annotations.
- **Memory sizes**. These refer to the memory of each input and output parameters for a given kernel. It has a direct correlation with the computational cost of kernel execution.

- **Execution time**. This refers to the amount of time a device takes to run a kernel for an input of certain size on a give device.

- **Data transfer time**. This refers to the amount of time the CPU takes to send and receive each input and output parameters for a given kernel from the CPU to a specific device and vice-versa.

The available devices can be obtained from the architectural description as defined in Section 3.2. The code annotations, as defined in Section 4.1 provide at least the list of available kernels. If available, the annotations can also be used to determine:

- The list of preferred devices for a kernel.

- An upper limit on memory usage for a given kernel. This can be used in conjunction with the previous information in order to filter the possible devices for a kernel.

- Input and output parameters to detect data dependencies between kernels.

The partitioning techniques will work with the tuple of kernel and input/output size. Each tuple takes a certain time to run, and has transfer rate for a specific device. Lastly, each device has its own strengths and limitations, and as such their performance will vary from kernel to kernel.

For the purposes of this Thesis, the combination of a kernel and a specific input size is named an *execution unit*. Thus, a given kernel will be considered a different execution unit when run with an input size of 256 or 512 bytes.

There are two important aspects to consider among execution units: *incompatibility* and *dependency*. Two execution units are *incompatible* if both cannot be executed together on the same application. The idea behind this is that the algorithm is able to consider different versions of the source code, selecting the best one for each device. For example, in a nested loop, it can decide whether to consider the outermost loop as a kernel that is run once, or the innermost loop as a kernel that is run once for each iteration of the outermost loop. One execution unit *depends* on another if it needs to wait until the latter has finished. This categorization allows to detect independent execution units, which can be run in parallel. Additionally, this allows differentiating between valid and invalid sequences of execution units, improving the computation of the best schedule accordingly.

Another measure of interest is the *feasibility* of deploying an execution unit on a specific device. One execution unit is *feasible* for one device if it can be executed on that device. This is required to decide which is the best device for each execution unit considering only valid devices for each execution unit. For example, any execution unit that performs system calls shall be considered unfeasible for a GPU device.

An execution unit with a sufficiently large input data can be unfeasible for a device with limited memory.

The previous concepts are formalized as follows:

Let $E$ be set of execution units $E = \{e_1, \ldots, e_l\}$, $D$ the set of devices $D = \{d_1, \ldots, d_n\}$ and $R$ the set of execution restrictions $R = \{r_1, \ldots, r_n\}$.

Let $r_e$ be the restrictions that apply to an execution unit, defined in Equation 5.1. Also, let $r_d$ be the restrictions that apply to a device, defined in Equation 5.2.

$$r_e =< e, R_e > | \; e \in E, R_e \subseteq R \tag{5.1}$$

$$r_d =< d, R_d > | \; d \in D, R_d \subseteq R \tag{5.2}$$

Let $R_e$ be the set of restrictions that apply to all the execution units $R_e = \{r_{e_1}, \ldots, r_{e_l}\}$, such that $|E| = |R_e|$, and $R_d$ be the set of restrictions that apply to all the devices $R_d = \{r_{d_1}, \ldots, r_{d_m}\}$, such that $|D| = |R_d|$

Given the previous sets, we consider the relationships of incompatibility, dependency, and feasibility, defined respectively in Equations 5.3, 5.4, and 5.5.

$$I \in M_{e \times e}(\{0,1\}) \mid I_{i,j} = \begin{cases} 1 \text{ if } e_i \text{ and } e_j \text{ are mutually exclusive} \\ 0 \text{ otherwise} \end{cases} \tag{5.3}$$

$$P \in M_{e \times e}(\{0,1\}) \mid P_{i,j} = \begin{cases} 1 \text{ if } e_i \text{ depends on } e_j \\ 0 \text{ otherwise} \end{cases} \tag{5.4}$$

$$F \in M_{e \times d}(\{0,1\}) \mid F_{i,j} = \begin{cases} 1 \text{ if } R_{e_i} \cap R_{d_j} = \emptyset \mid R_{e_i} \in R_e, R_{d_j} \in R_d \\ 0 \text{ otherwise} \end{cases} \tag{5.5}$$

The sets and constraints defined so far are extended to include data partitions. Execution units can be partitioned in order to execute different parts simultaneously on at least two devices. Such partitions share the kernel code, but have only a fraction of the input and output datasets. Consequently, data can be processed in parallel. Splitting execution kernels also permits to run execution units with bigger input/output data sets than a device can handle. Each partition is responsible for transferring the portion of the required input/output data to and from the device. Thus, kernel code must be extended to include data slicing and additional transfers. However, not all cases can be partitioned. In general, partitioning is possible in most scenarios where each execution thread has no data dependencies on other threads, which is the typical case of kernels that can be mapped to SIMD computing devices. The worst case scenario arises in the case of data inputs that cannot be partitioned as the whole input dataset must be used by every task. In this case, the whole input

needs to be sent to each partition device. Even when possible, partitioning is not always the most efficient solution. There are several cases when this is the case, such as those execution units with small workloads or input sizes, algorithm where the computational time is reduced compared to the transfer time, etc. Lastly, a threshold will be considered before partitioning. The purpose of this threshold is to avoid "*false positives*", where the algorithm proposes a partition that would yield a very small speedup, and because of the variance of the measures, results would be worse than not partitioning at all. Through a preliminary testing process, this threshold has been established to 10% of the data size for all cases. Whenever less than that amount is proposed to be moved to a device other than the CPU, the partition is ignored and everything is run in the CPU.

A wide range of possible execution unit partitions is considered: given the sets of execution units $E$ and devices $D$, a set of execution unit partitions $EP = \{ep_1, \ldots, ep_r\}$ is defined. Each execution unit partition $ep_i = (e_a, d_b, g_c, r_d)$ consists of an execution unit $e_a$, a device $d_b$, a range $g_c$ and a ratio $r_d$. The range $(g_c)$ represents each of the executions of the kernel code when the data set does not fit into the device memory and, consequently, several executions are needed for different subsets of the data set. For example, a 4 GiB dataset run in a device with a memory capacity of 2 GiB would lead to two ranges. This is an instance of the memory bound problem. The algorithm must ensure that two execution unit partitions with the same execution unit, device, and range are not included in the same scheduling plan. The ratio $(r_d)$ is the fraction of the input dataset that is assigned to an execution unit partition. For example, if the input data set is divided into 20% to the CPU, 50% to a GPU and 30% to another GPU, corresponding ratios of partitions would be 0.2, 0.5, and 0.3. A wide set of rates from 0 to the maximum rate that can be stored in the device memory are considered for each range.

After creating the partitions set, the algorithm has to obtain a scheduling plan composed by a sorted list of execution unit partitions
$SP = \{ep_1, \ldots, ep_m\}$. The plan should cover all the kernels required with their complete datasets and achieve the best possible performance. The incompatibility, dependency and feasibility matrices are extended to cover execution unit partitions $(IP \supseteq I, PP \supseteq P, FP \supseteq F)$.

Lastly, a function to estimate the execution time of each kernel partition is required. This function relies on profiling data from previous kernel executions on those devices with different input data sizes. The algorithm requires execution time for at least two data sizes, and the transfer times from the host to all devices for at least two data sizes (not necessarily the same ones). The database does not need to be extensive, but it requires the execution and transfer times for the largest and the smallest data sizes, so that the algorithm may apply linear interpolation. In order to test the algorithm, execution times of several use cases has been taken, with several input data sizes for each use case. Furthermore, data transfer times are stored for the different devices considered in the scheduling process.

A simple linear interpolation function is used taking profiling data to provide estimations for data sizes that are yet to be measured. As for the partitions, the function allows to estimate the execution and transfer times for partitions different to the ones that have been measured. This allows to provide a relatively high speedup with a small profiling work. The interpolation method is supposed to be incremental (only for applications that are meant to be executed more than once). The first time the interpolation is done with few values and a great margin of error. As executions are being made more values are obtained and the estimations get better.

The smallest and biggest limits for the interpolation also change as the application is executed several times. Initially, the smallest and biggest values are those of the smallest and biggest input datasets provided by each of the benchmarks used in this work. However, in a general case without such information, an application may start with two arbitrary sizes. As kernels are partitioned and executed, feedback provides a path to improve accuracy.

Although we use here this simple estimation function, more complex functions may be used. For example, it could be of interest to provide estimations based on code characteristics (via fuzzy logic, genetic algorithms or similar techniques), rather than using simple linear interpolation.

To ensure that when a kernel is started all the needed input data set is available at that device, the location of every parameter needs to be tracked. To this end, a list of parameter locations is kept and updated, representing the computing devices where these parameters are located. Before executing a new execution unit partition, all the input parameters that are not already on the device memory space where the partition is to be run need to be be transferred.

Our model does not take into consideration the possibility of device-to-device transfers. Therefore, all the parameters needed by a device are first looked up in the host. In the event that the parameters are not found in the host, but have been produced in a different device, they are first transferred from this device to the host, and secondly, all the needed parameters are transferred from the host to the requesting device. Whether data will be completely transferred or only partially transferred depends on kernel partition rate.

Similarly, output parameters need to be taken into consideration. Complete output parameters are left on the device while partial output parameters should be transferred to the host to be combined with the rest of parameter data. A partial parameter mapping is used to state which parameters are partially stored on the host and their corresponding storage rate. This mapping is updated each time that partial output parameters are transferred to the host. If one partial parameter is completed then it is moved from this mapping to the parameter location list.

When the parameter location list is initialized, the parameters available at start-up are stored on the host. Any output parameter that needs to be produced is unchecked until the time where an execution unit creates it. The list of parameter locations is updated after each kernel partition execution. At the end, the final

output parameters need to be transferred from the device where they are stored to the host.

Once the scheduling plan is completed, it can be executed using an execution algorithm. This algorithm selects the next execution unit partition to be executed on each device. Then it runs the execution unit and performs the data transfers in the background. Finally, it waits until the current execution unit partition and its transfers are finished to select and run the next one.

The whole process is divided into three phases: the partitioning phase, the scheduling phase and the execution phase. Figure 5.1 shows the general work-flow. There is a preliminary process where the architectural description and the code annotations are processed in order to generate the setup files. These files contain the following information:

- Number of sizes to consider when partitioning.
- Compatibility and dependency matrices.
- Viability matrix, showing whether or not a device can execute a kernel.
- Measures for different data sizes. Used in order to predict the results with a new size.
- Measures for data transfers of different sizes.

This information can be obtained automatically from the architectural description and the code annotations. However, for this Thesis, the setup has been generated manually.

The static partitioning engine proper has three stages. Stage one takes an input source code and generates the full list of execution units and partitions to be considered. This list is passed to the scheduling phase, which in turn will find the theoretical best scheduling plan. Afterwards, this plan will be executed in the third phase. The second phase is the longest out of the three, mainly because it has to generate many possible final schedules and even more partial schedules. In the case that there is a need to decrease the execution time of this phase, the list generated in the previous phase should contain as few partitions as possible. The trade-off would be the worsening of the performance that the schedule plan may attain.

Regarding the variable parameters, the number of devices can be disregarded. The reason for this is that the total number of devices available in the platform will hardly exceed four and, even then, most kernels will only be suited for one or two of them. On the other hand, the kernels, input data sizes and partitions can be considered together as execution units. Based on the number of execution units, the complexity of the first stage is $O(n^3)$. The complexity of the second stage is, at worst, $O(n^4)$, and in the general case will be $O(n^3)$. The last stage depends entirely on the complexity of the executed partitions, so it has no fixed complexity.

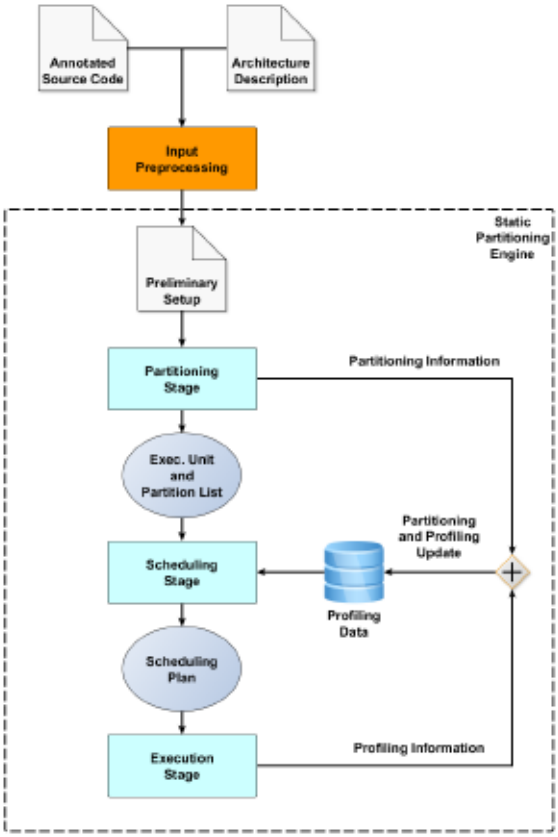


Figure 5.1: Algorithm workflow.

However, at the very least, the complexity would be O(n), assuming all the executed partitions are of complexity O(1).

### 5.1.1 Partition phase

The procedure for the partition phase is outlined as follows:

1. The set of execution partitions $EP = \{ep_1, \ldots, kp_r\}$ is defined for each valid combination of execution unit, device, range and rate: $ep_i = (e_a, d_b, g_c, r_d)$

2. For each feasible tuple of execution unit and device, at least one range is defined. The first range is associated with a set of partial rates from 0 to the maximum rate that can be stored in the device memory (called memory bound of the device). If this maximum rate is not equal to 1, then several other ranges should be created with just two rates associated (0 and the memory bound of the device). The remaining data will be split according to the number of possible rates. The number of ranges should be enough to execute the whole dataset with one partition from each range. Details from this step are

provided in Algorithm 2. First the number of necessary ranges is computed. Afterwards, the rates are calculated as explained before. For example, in a two range dataset of 60 and 40, the first range will have two rates (0 or 60), whereas the second will be split according to the number of possible rates. Lastly, the partitions are established according to the number of ranges and rates.

3. An extended incompatibility matrix is created for partitions. Two partitions are incompatible if their execution units are incompatible or their kernels, devices and ranges are the same.

4. An extended dependency matrix is created for partitions. One partition depends on another if its execution unit also depends on the kernel from the other partition.

### 5.1.2 Scheduling phase

Algorithms 3 through 8 show the scheduling algorithm and the associated auxiliary procedures. They can be outlined as follows:

1. A search tree is created to represent all the possible scheduling plans. The root node represents the original, empty scheduling plan. Intermediate nodes are partial scheduling plans. Lastly, the leaf nodes are complete scheduling plans. Each plan is associated with an execution time estimation. The parameter location array of the root node is initialized with initial parameters stored on the host.

2. A child node with a new plan is created by adding an execution unit or a partition to the scheduling plan of the parent node. Each parent node can have a child node for each execution unit or partition that:

   - Is not included in the parent node plan.
   - Is not incompatible with any other kernel partition within the parent scheduling plan.
   - Does not depend on any other partition that is not included in or incompatible with the parent plan.

   If there is none, the scheduling is complete and this is a leaf node.

3. When a new kernel partition is added, the following transfers should be obtained:

   - The transfers of all the input parameters from whatever device they are stored to the host.

**Algorithm 2** Partitioning Phase.

1: let *numRanges* be the number of ranges required to cover the full data size of an execution unit in a device.
2: let *MaxNumOfRates* store either the number of rates associated with an execution unit. If the execution unit fits in a device in a single range, this variable will be the number of possible rates. If not, then it will store the rates for the last, smallest range.
3:
4: **procedure** EXECUTIONUNITPARTITIONALGORITHM(*EP*,*numEP*, *NumPossibleRates*)
5: $numEP \leftarrow 0$
6: **for** i in 1..*numExecutionUnits* **do**
7: **for** j in 1..*numDevices* **do**
8: **if** $F_{i,j} = 1$ **then**
$numRanges \leftarrow \lceil$ *data size of execution unit* $e_i$ / *memory bound of device* $d_j \rceil$
9: **if** $numRanges > 1$ **then**
$MaxNumOfRates \leftarrow$ *memory bound of device* $d_j$ / (*data size of execution unit* $e_i$ / *NumPossibleRates*)
10: **else**
11: $MaxNumOfRates \leftarrow NumPossibleRates$
12: **for** k in 1..*numRanges* **do**
13: **if** $k = 1$ **then**
14: **for** l in 0..*MaxNumOfRates* **do**
15: $rate = (l/NumPossibleRates)$
16: $EP_{numEP} = (i,j,k,rate)$
17: $numEP = numEP + 1$
18: **else**
19: $EP_{numEP} = (i,j,k,0)$
$EP_{numEP+1} = (i,j,k,\ MaxNumOfRates/NumPossibleRates)$
20: $numEP = numEP + 2$

- The transfers of all the input parameters that are completely required, from the host to the selected device.
- The transfers of all the input parameters that are partially required, from the host to the selected device.
- The transfer of all the outputs that are partially generated from the selected device to the host.

4. The parameter location array and the partial parameters vector are updated accordingly. The transfer time required for those parameters is added to the

**Algorithm 3** Scheduling Phase.

```
1: let bestTime store the best execution time of a plan at any given moment.
2: let bestSP store the best scheduling plan at any given moment.
3: let queue store newNodes, which contain partial scheduling plans, except for lead
   nodes.
4: let ListNextEP list the next execution units elegible to be run after the point in
   time scheduled in node
5:
6: procedure SCHEDULINGALGORITHM(bestSP, bestTime)
7:     bestTime ← MAX
8:     queue ← INSERT(queue, initialSched)
9:     while not EMPTY(queue) do
10:        node ← EXTRACT(queue)
11:        ListNextEP ← listNextExecutionUnitParts(node)
12:        for i in ListNextEP do
13:            newNode ← ADDEP(node, EP_i)
14:            if isValidSched(newNode) then
15:                if not isFinalSched(newNode) then
16:                    queue ← INSERT(queue, newNode)
17:                else
18:                    if bestTime>getTime(newNode) then
19:                        bestSP ← SCHEDPLAN(newNode)
20:                        bestTime ← getTime(newNode)
```

execution time of the kernel partition.

5. The execution time associated to each node is equal to the finishing time for the latest partition plus the transfer time for the final output transfer operation. The finishing time of a new partition is computed by adding the estimated execution/transfer time of the new partition to the maximum of: 1) the finishing time of the latest partition executed on the same device. 2) the finishing time of the latest partition on which the new kernel partition depends. The time of the final transfer operation is the time required to send the final parameters from whatever device they are stored to the host.

6. Any scheduling plan where the added rates of the partitions with the same kernel is bigger than 1 is invalid, and its node must be discarded. Furthermore, if the partitions cover all the possible ranges of this kernel and the added rate is lower than 1, the plan is invalid and its node, too, must be discarded.

7. The scheduling plan with the lowest execution time will be selected for execution.

It is important to note that, as far a scheduling goes, there is very little difference between execution units and partitions. While it is true that the use of partitions allows to bypass restrictions such as the memory bounds of a device, in the end they are treated as smaller execution units that inherit the dependencies and incompatibilities of the execution unit that was split, and are also incompatible with said execution unit. In this way, the choice of splitting an execution unit, or running it completely, requires no special cases within the algorithm. In fact, an execution unit can be considered as a partition with a rate of 1 for the purpose of our algorithm.

Figure 5.2 shows an example of the scheduling algorithm for a very simple scenario. In it, we want to schedule two kernels, *K0* and *K1*, and have two devices available, CPU and GPU. The root node, node 1 in the figure, represents the empty schedule, where no kernel has been assigned to a device yet. From there, we expand in depth. The next node (node 2) shows the first kernel, *K0*, assigned to the first device, CPU, for a cost of 5 units. Next, we assign the next kernel, *K1*, to the first device, CPU, for a total cost of 7 units (node 3). As this is a complete schedule, we mark it as the current best and backtrack for the next schedule. According to the algorithm, next schedule, node 4, carries on from node 2, and assigns *K1* to the next device, GPU, for a total cost of 11. Since the total cost is higher than that of the current best, we discard the schedule, and keep backtracking. Next schedule, node 5, carries on from node 0. We assign *K0* to the next device, GPU, for a total cost of 8 units. Again, the cost is higher than that of the current best, so we discard the whole branch. Since there are no more options to explore, we have the final schedule (node 3).

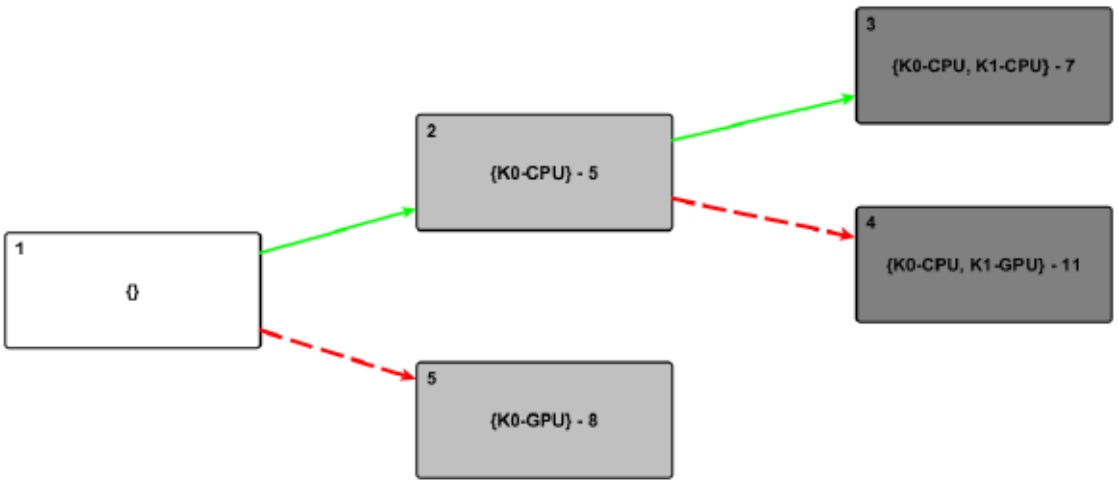


Figure 5.2: Example of scheduling algorithm.

### 5.1.3 Execution phase

The execution phase is the stage of the algorithm when the application is run. Algorithm 9 shows the procedure. The algorithm for the execution phase can be outlined as follows:

1. The execution of the execution units and/or partitions with their input and output transfers is done in background. A finalization event raises when it is done and an associated handle is executed.

2. The handler function is executed whenever an OpenCL finalization event raises. Firstly, it obtains which partition was finalized. Then, it audits all the free devices and check which one is the next execution unit or partition to be executed on each one. If these only depend on other execution units or partitions that have already finished, they are executed in background and the same handle is associated to the finalization event.

3. When the last execution unit or partition is completed, the handler will finish the execution.

4. The application begins executing the handler function directly in order to launch one task per device (if possible).

For the implementation of this phase we have used the OpenCL API. For example, the finalization event raised will be an OpenCL finalization event.

**Algorithm 9** Execution Phase.

1: let $EP_r$ store the next partition scheduled for a given device through procedure *nextEP*.
2: let $ListDep_{EP_r}$ store the list of dependencies for partition r that have yet to be met.
3:
4: let procedure *FINISH* check whether or not a device is done executing a partition.
5: let procedure *EXTRACT* remove the finished partition from the scheduling plan.
6: let procedures *INPUTTRANSFERS* and *OUTPUTTRANSFERS* transfer data to and from a device, respectively.
7: let procedure *EXEC* launch a partition in a device.
8:
9: **procedure** SCHEDULER()
10: HANDLER()
11: **while** $schedPlan \neq \phi$ **do**
12: HANDLER()
13: **procedure** HANDLER()
14: **for** i in 1..$numDevices$ **do**
15: **if** FINISH($device_i$) **then**
16: EXTRACT($schedPlan$, $device_i.EP$)
17: $EP_r \leftarrow$ NEXTEP($schedPlan$, $device_i$)
18: $ListDep_{EP_r} \leftarrow \forall_{EP_j \in schedPlan} \| (ep_r, ep_j) \in R_{PP}$
19: **if** $ListDep_{EP_r} = \phi$ **then**
20: INPUTTRANSFERS($EP_r$, $device_i$)
21: EXEC($EP_r$, $device_i$)
22: OUTPUTTRANSFERS($EP_r$, $device_i$)
23: $device_i.EP \leftarrow EP_r$

## 5.2 Summary

In this chapter we have proposed a set of task scheduling techniques to find the best orchestration of a set of kernels for a given architecture. This techniques rely on the architecture description and code annotation to find the best schedule based on feedback. In the following chapter, we will show the evaluation of the different proposals made in this Thesis.

**Algorithm 4** Scheduling Phase: Auxiliary functions 1.

1: let *usedEP* store those partitions already used in the scheduling plan stored in *node*.
2: let *IncompEP* store those partitions that are incompatible with the scheduling plan stored in *node*.
3: let *RemainEP* store those partitions that can still be used after the point in the scheduling plan stored in *node*.
4: let *RedPP* store the list of dependencies between remaining partitions.
5: let *IndepEP* store those thos partitions that are have no dependencies.
6:
7: **function** LISTNEXTEXECUTIONUNITPARTS (node)
8: $usedEP \leftarrow usedEP_i = 1 \Leftrightarrow EP_i \in$ SCHEDPLAN($node$)
9: $IncompEP \leftarrow IncompEP_i = IP_{i,0} \wedge usedEP_0 \vee \cdots \vee IP_{i,n} \wedge usedEP_n$
10: $RemainEP \leftarrow RemainEP_i = \neg(IncompEP_i \vee usedEP_i)$
11: $RedPP \leftarrow RedPP_{i,j} = PP_{i,j} \wedge RemainEP_i \wedge RemainEP_j$
12: $IndepEP \leftarrow IndepEP_i = \neg(RedPP_{i,0} \vee \cdots \vee RedPP_{i,n})$
13: $ListNextEP \leftarrow ListNextEP_i = RemainEP_i \wedge IndepEP_i$
14: **return** *ListNextEP*
15: **function** ISVALIDSCHED(node)
16: **for** $EP_i$ in SCHEDPLAN($node$) **do**
17: $(execution\ unit, device, range, rate) \leftarrow EP_i$
18: $usedRanges_{execution\ unit, device} \leftarrow usedRanges_{execution\ unit, device} + 1$
19: $usedRates_{execution\ unit} \leftarrow usedRates_{execution\ unit} + rate$
20: $Overflow \leftarrow \exists_i \parallel (usedRates_i > 1)$
21: $Underflow \leftarrow \exists_i \forall_j \parallel (usedRates_i < 1) \wedge (usedRanges_{i,j} = \lceil data\ size_i / memory\ bound_j \rceil)$
22: **return** $\neg(Overflow \vee Underflow)$
23: **function** ISFINALSCHED(node)
24: $usedEP \leftarrow usedEP_i = 1 \Leftrightarrow EP_i \in$ SCHEDPLAN($node$)
25: $IncompEP \leftarrow IncompEP_i = IP_{i,0} \wedge usedEP_0 \vee \cdots \vee IP_{i,n} \wedge usedEP_n$
26: $RemainEP \leftarrow RemainEP_i = \neg(IncompEP_i \vee usedEP_i)$
27: **return** $\forall_i \parallel RemainEP_i = 0$

**Algorithm 5** Scheduling Phase: Auxiliary functions 2.

1:
2: let *tExec* store the amount of time it would take to transfer input and output data, as well as execute, a given partition.
3: let *tEndDepEP* store the times it would take to meet each of a partition's dependencies.
4: let *tStartEP* store the worst-case starting time for a partition.
5:
6: **function** GETTIME(node)
7: **for** $i$ in SCHEDPLAN($node$) **do**
8: $tExec \leftarrow executionTime_{EP_i}$
9: $tExec \leftarrow tExec +$ GETTRANSFERTIME($EP_i$)
10: $tEndDepEP \leftarrow tEndDepEP_r = PP_{i,r} \cdot tEndEP_r$
11: $tStartEP_i = max(tEndDev_{EP_i.device}, max_r(tEndDepEP_r))$
12: $tEndEP_i = tStartEP_i + tExec$
13: $tEndDev_{EP_i.device} = tEndEP_i$
14: **return** $max_{dev}(tEndDev_{dev})$ + GETFINALTRANSFERTIME()

**Algorithm 6** Scheduling Phase: Transfer Auxiliary Functions 1.

1: This procedure calculates the transfer time necessary to move input and output data from one step to the next.
2: **procedure** GETTRANSFERTIME($EP_k$)
3: $(execution\ unit, device, range, rate) \leftarrow EP_k$
4: TRANSFERSPERUNITEXEC($EP_k$)
5: UPDATEPARAMLOCATION($EP_k$)
6: **for** i **do** in 1..$numDevices$
7: $aux \leftarrow aux_i = 1 \Leftrightarrow CompTransfAnyHost_i == i$
8: $time = time +$ TRANSFERTIME($i, aux$)
9: $time = time +$ TRANSFERTIME($device, compTransfHostDev$)
10: $time = time +$ TRANSFERTIME($device, rateTransfHostDev$)
11: $time = time +$ TRANSFERTIME($device, rateTransfDevHost$)
12: **return** $time$
13:
14: This procedure calculates just the transfer time to gather the output data in the host at the last stage.
15: **procedure** GETFINALTRANSFERTIME()
16: $aux \leftarrow aux_i = FinalHostParams_i - (paramLocation_{host,i} \wedge FinalHostParams_i)$
17: **for** j **do** in 1..$numDevices$
18: $paramsInDev \leftarrow paramsInDev_i = aux_i \wedge paramLocation_{j,i}$
19: $aux \leftarrow aux_i = aux_i - paramsInDev_i$
20: $CompFinalTransf \leftarrow CompFinalTransf_i = CompFinalTransf_i + (paramsInDev_i * i)$
21: $time = time +$ TRANSFERTIME($j, paramsInDev$)
22: **return** $time$

**Algorithm 7** Scheduling Phase: Transfer Auxiliary Functions 2.

This procedures calculates the devices where the input and output variables must reside in, in order to execute the next step.

**procedure** TRANSFERSPERUNITEXEC($EP_k$)
  $(execution\ unit, device, range, rate) \leftarrow EP_k$
  **if** $rate = 0$ **then**
    **return** 0
  $Inputs \leftarrow Inputs_i = CompInputs_i \vee RateInputs_i$
  $InputsReady \leftarrow InputsReady_i = paramLocation_{device,i} \wedge Inputs_i$
  $InputsMove \leftarrow InputsMove_i = inputs_i - InputsReady_i$
  **if** $rate = 1$ **then**
    $compTransfHostDev_{EP_k} \leftarrow InputsMove$
  **else**
    $compTransfHostDev_{EP_k} \leftarrow compTransfHostDev_{EP_k,i} = CompInputs_i \wedge InputsMove_i$
    $rateTransfHostDev_{EP_k} \leftarrow rateTransfHostDev_{EP_k,i} = (RateInputs_i \wedge InputsMove_i) * rate$
    $rateTransfDevHost_{EP_k} \leftarrow rateTransfDevHost_{EP_k,i} = RateOutputs_i * rate$
  $InputsMoveToHost \leftarrow InputsMoveToHost_i = InputsMove_i - (paramLocation_{host,i} \wedge InputsMove_i)$
  $aux \leftarrow InputsMoveToHost$
  **for** j in 1..$numDevices$ **do**
    $paramsInDev \leftarrow paramsInDev_i = aux_i \wedge paramLocation_{j,i}$
    $aux \leftarrow aux_i = aux_i - paramsInDev_i$
    $CompTransfAnyHost \leftarrow CompTransfAnyHost_i = CompTransfAnyHost_i + (paramsInDev_i * i)$

**Algorithm 8** Scheduling Phase: Transfer Auxiliary Functions 3.

1: This procedure updates the last known location of the input and output parameters.
2: **procedure** UPDATEPARAMLOCATION($EP_k$)
3: $(execution\ unit, device, range, rate) \leftarrow EP_k$
4: $paramLocation_{host} \leftarrow paramLocation_{host,i} = paramLocation_{host,i} \vee InputsMoveToHost_i$
5: $paramLocation_{device} \leftarrow paramLocation_{device,i} \vee compTransfHostDev_i \vee CompOutputs_i$
6: **if** $rate = 1$ **then**
7: $paramLocation_{device} \leftarrow paramLocation_{device,i} \vee RateOutputs_i$
8: $rateHostParameters \leftarrow rateHostParameters_i + rateTransfDevHost_i$
9: $completedParams \leftarrow completedParams_i = 1 \Leftrightarrow rateHostParameters_i == 1$
10: $rateHostParameters \leftarrow rateHostParameters_i - completedParams_i$
11: $paramLocation_{host} \leftarrow paramLocation_{host,i} + completedParams_i$

# Chapter 6

# Evaluation

This chapter shows the evaluation of the different components of the proposed framework. Of the three pieces of software described in Figure 1.1, only the automatic code annotation and the partitioning techniques are included in this Chapter. The reason for this is the automatic hardware extraction is a straightforward, auxiliary tool that simply has to gather certain information from various tools and compose the full architecture description.

## 6.1 Reference platform

The reference platform for all tests is described in Table 6.1. AMD driver [11] was used instead of Intel OpenCL driver because the latter produced results with a big variance, as opposed to the AMD's more stable results. The operating system used during the tests is Ubuntu 14.04.2 LTS. The code has been compiled with g++ 5.1.2. The developed code has been compiled with the flags *-std=c++11 -O3*. For the benchmarks the standard configuration has been used in each case, and no optimization has been used for the CPU code. This has been done to ensure that all devices can be evaluated on equal ground.

## 6.2 Evaluation of the software annotation techniques

In order to evaluate AKI, we have applied it to some of the benchmarks in the Parboil benchmark set [114]. The idea is to preserve as much of the original source code as possible. For this, the simplest option is to use AKI with the base, sequential benchmarks and compare them with the OpenMP equivalent, which can be found in Parboil. We have evaluated four benchmarks, which have only one source code file using OpenMP annotations.

Table 6.1: Test platform.

| | Intel CPU | AMD GPU | Intel Xeon Phi |
|---|---|---|---|
| Model | Intel®Xeon® CPU E5-2695 | AMD®Radeon® R9 290 series | Xeon Phi® coprocessor 3120 series |
| Core clock | 1.2 GHz | 1.0 GHz | 1.1 GHz |
| Computing units | 24 | 2816 SP | 224 |
| Memory | 128 GiB | 4 GiB | 6 GiB |
| OCL driver | AMD-APP-SDK-v2.9 | Runtime 14.2 | OpenCL™ |
| OCL version | 1.2 | 2.0 | 1.2 |

### 6.2.1 Benchmarks

In order to evaluate AKI, we have employed four benchmarks: Breadth-first search, histogram, Lattize-Boltzman Method Simulation (LBM), and matrix multiplication (SGEMM).

The BFS [76] is an algorithm used to find the shortest path trough tree- or graph-like structures. It starts at the top of the structure, and moves to the bottom by exploring all neighbour nodes of a level at a time, hence the name. It is an alternative to other methods, such as the depth-first search, or other search algorithms.

The histogram benchmark is a simple accumulation of values into a vector. In the case of the Parboil histogram, the values are part of a two-dimensional matrix. The bottleneck of this benchmark is the accumulation (or reduction) of values from the matrix to the histogram vector. Parboil detects the region of the vector where most conflict is expected, which is typically in the middle of the histogram, and assigns workers to make partial reductions of this area before doing a last accumulation.

The LBM [34] benchmark is a form of fluid simulation consisting on the solving of systems of differential equations. It has recently been used in benchmarking as an alternative to other algorithms such as the S3D benchmark found in SHOC [39], which is instead based on Navier-Stokes equations.

The SGEMM is an example of matrix multiplication. It is a problem involving two matrices, of size XxY and YxZ, where X, Y and Z may be any number. The algorithm takes all the rows of the first matrix, and multiplies them, element by element, with all the columns of the second matrix (hence the same dimensions). There are several optimizations of this algorithm, such as the Strassen algorithm [113], or transposing the second matrix to optimize cache access.

### 6.2.2 Results

The results can be seen in Table 6.2. The columns represent each of the benchmarks that have been used for evaluation. For each benchmark, we show the lines that correspond with for loops in the original source code meeting the following criteria: for loops detected by AKI during the hotspot selection, for loops detected as potential kernels by AKI during the kernel selection, original kernels annotated in OpenMP (baseline), OpenMP kernels that have not been executed and therefore have not been annotated, OpenMP kernels that are in a nested loop and have not been annotated, and lastly, encountered failures. These refer to those for loops that have not been identified, but do not fall into any of the previous two criteria (i.e. not executed or nested). Given the results, we observe that AKI detects most of the potential kernels. As explained avobe, we consider three different categories of failures. The first category includes those kernels that have an OpenMP directive in the benchmark, but are not executed. In these cases, our algorithm will not detect them, since they have no computational load. For this reason, no corrective action will be taken to detect and annotate them. The second category includes nested loops that have been deliberately ignored. Since a decision was made to only annotate the innermost loops in case there was a choice, AKI will not detect other loops, not because of the algorithm, but by design. The nested loops will change in future versions of the algorithm, according to the parametrized granularity that the user desires, so while matching in these cases can and will be improved; they are not the main concern. Lastly, the failures that do not fit into any of the previous categories need to be studied closely. Of these, the missed loop in the histogram benchmark (line 98) is a false negative. The simplicity of the loop makes it look like an initialization loop, and so it is missed as a kernel. Further fine-tuning of the thresholds or feedback from the user can prevent this case. Lastly, the three loops in LBM that are not considered as the initial OpenMP annotated kernels have dependencies between iterations and write into global variables. Controlling these dependencies would be costly and, since our algorithm does not take them into consideration yet, they are annotated as AKI kernels.

## 6.3 Evaluation of the static partitioning algorithm

The proposed partitioning algorithm has been evaluated separately from the annotation techniques, using various use cases. They are pieces of OpenCL code calling one or more kernels. In this experiment, the number of kernels ranges from one to twenty-seven. They have been executed on the architecture described in the reference platform. The selected examples, which are listed below, are considered to be representative from many kernel-based programs. Experimental results show the improvement obtained with the proposed scheduling as opposed to more typical solutions, such as running the program in a single device, or not partitioning the input

Table 6.2: Parboil benchmark evaluation comparing AKI with manual OpenMP annotation. The numbers represent the lines of the code in which a kernel is annotated

| | BFS | Histogram | LBM | SGEMM |
|---|---|---|---|---|
| Hotspots | 86,161,175 | 98,101 | 62,103,104,127, 128,186,459,562,563 | 28,29,30 |
| AKI | 86 | 101 | 62,127,128,186, 459,562,563 | 28,29,30 |
| Baseline | 86 | 98,101 | 62,126,157, 186,289,348 | 28 |
| Not executed | - | - | 157,289,348 | - |
| Nested | - | - | 126 | 29,30 |
| Failures | - | 98 | 459,562,563 | - |

and output data. The following benchmarks have been extracted from the Rodinia benchmark [32, 33], the SHOC benchmark [39], the Intel Code OpenCL Samples [63] or have been developed ad-hoc. The list of benchmarks as well as a description of them can be found in Section 6.3.1.

The implementation of these benchmarks is based on a core code composed of one or several kernels that are executed once or several times within a loop. This is the way most of the state-of-the art benchmarks for kernel-based programs are implemented. Our approach on those loop-based algorithms is to focus on scheduling one iteration in the best possible way. Then, each iteration is executed using the selected scheduling. This approach compromises between obtaining a good scheduling performance and reducing the problem complexity to get the scheduling plan on a reasonable time. Furthermore, in many cases, each iteration depends completely on the results of the previous iteration, and consequently each iteration can be analysed as a separated scheduling problem. Those applications need to have an implementation in OpenCL or other kernel-based framework. It is for this reason that these benchmarks have been chosen, since it saves the effort of coding new versions from scratch.

### 6.3.1 Benchmarks

**The stencil benchmark** used in this work is adapted from the 3D stencil found in the Rodinia benchmark [32, 33]. It takes a matrix of any dimensions, and updates

each element of the matrix (typically, a pixel in an image) based on the values of the neighbours of the element. There are many variants of the convolution, based on the precise filter that is to be applied. The concurrency can be more or less relaxed, based on whether or not the elements must be updated in a particular order.

The program can be partitioned amongst the devices that make up the heterogeneous platform. The benchmark has a single kernel that performs the stencil algorithm. The partitioning can be achieved by splitting the input matrix at the row level. This also applies to the output matrix. In the case of partitioning, the host will send the input partitions, and receive the output ones.

**The transitive closure benchmark** implements the transitive closure of a binary relationship among the elements from a set. The relationship is defined using a binary square matrix. The algorithm is a pipeline computation, where each iteration performs three steps: 1) multiply the original matrix and a derived matrix, 2) check whether the matrix is the transitive closure and if not 3) update the current matrix by adding it to the original matrix. The last two stages depend on the first, but can be run in parallel as an optimization. There are more optimal versions of this algorithm, such as the Floyd-Warshall algorithm [49].

For the purpose of this test, we will only run one iteration of the pipeline, and assume that stages two and three are the ones from the previous iteration. Therefore, the three operations are implemented as independent kernels that can be executed in parallel on each iteration. A previous study shows that the multiplication kernel is the one that takes the longest time to run. For this reason, this kernel can be partitioned across the devices, whereas the other two are executed completely on one device. The input of the multiplication kernel is split by fragmenting all the matrices at the row level. To allow this, it is necessary that the second input matrix is transposed before calling the kernel.

**The S3D benchmark** is an OpenCL/CUDA implementation of the Navier-Stokes equations [125] for a regular 3D domain, used to simulate turbulent combustion [60]. As explained before, it is an alternative to the equations used in the LBM benchmark.

The benchmark is composed of 27 kernels that have several dependencies between them. Most of the computation is held by two groups of parallel kernels (8 kernel each) that are, in fact, a manual partition of two bigger kernels. Because of this reason, there is no point on performing any more partitioning. Each kernel is executed completely on one device.

**The LU Decomposition benchmark** is used directly as it is implemented in the Rodinia benchmark suite. It is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix, and has many dependencies between threads, which means it is a complicated benchmark to optimize. The version available in Rodinia is composed of two sequential kernels that run in a loop, and suffer huge performance penalties due to the use of synchronization barriers

between the threads.

**An optimized General Matrix Multiply (GEMM) benchmark** is included in order to use efficiently the internal hardware characteristics of the Intel(R) Xeon Phi(tm). It is extracted from the Intel OpenCL Code Samples. The algorithm is an optimization over the one explained in Section 6.2. It optimizes the trivial matrix multiplication nested loop version to utilize the memory cache more efficiently by introducing a well-known practice as tiling (or blocking), where matrices are divided into blocks, and the blocks are multiplied separately to maintain better data locality.

**Monte Carlo simulation**. This benchmark simulates European stock option pricing through Monte Carlo algorithm of random sampling [85]. The option price calculation is performed using Black-Scholes stock pricing model [23] which, incidentally, is one of the benchmarks in the Parsec benchmark suite [22]. The implementation of the Monte Carlo simulation provided in the OpenCL samples consists on a single kernel with a huge computational load, where each threads repeats the following steps: 1) generate the random numbers, 2) calculate the stock price and 3) accumulate the calculated value. After this process is done, an average is calculated to obtain the final option price. There is no inter-dependence in this algorithm, which makes it possible to run all threads without any synchronization mechanism.

**The median filter** is a non-linear digital filtering technique used in order to reduce noise. Not unlike the stencil, it is a neighbour-related problem, where the algorithm goes through every element, and replaces it with the median of all neighbours. It is possible to apply this algorithm to any-dimensional data structures. Because the bulk of this algorithm falls on the median, most efforts have been put towards improving the updating of the values. The median filter found in the Intel OpenCL samples again consists on a single kernel where every thread computes the filter for a single element. Although there is some inter-dependence, given that all elements read the neighbouring ones, the problem is relaxed so as to avoid the use of synchronization mechanisms.

**The bitonic sort** is a parallel algorithm for sorting. It is also used as a construction method for building a sorting network. It is part of a family of algorithms called oblivious, because they behave in the same way regardless of the input. This makes it a great algorithm to run on GPU [100]. The idea behind this algorithm is to, at certain points, do one-on-one comparisons between the elements introduced as input. If they are ordered, they are left as they are. Otherwise, they are swapped. Because this comparisons are completely independent from each other, the degree of parallelism is high. The version of the bitonic sort provided with the Intel OpenCL samples takes inputs of four elements, and performs three swaps with them. Again, much like the previous two, this implementation has a single kernel.

### 6.3.2 Results

The evaluation process for the stencil and the transitive closure benchmarks is performed as follows. Firstly, profiling data is gathered for each kernel from the benchmark. Each kernel is executed independently on each device using inputs of different sizes. This profiling information will be used to interpolate the expected performance for each kernel with different input sizes and different levels of partitioning. Also the information about the incompatibility, dependency and feasibility of each kernel is gathered. Secondly, the proposed algorithm is executed for a selected problem size and partition pattern. The resulting scheduling plan is implemented on the benchmark and executed to obtain its performance. Finally, all the possible schedules considered in the algorithm are also executed and their performance results are compared to the selected solution.

The evaluation process for the S3D benchmark is almost the same as the former ones. The only difference is that the selected scheduling plan cannot be compared to all the other schedules considered because the amount of tests would make it infeasible ($2^{27}$ combinations). Therefore, we compare the chosen schedule with the original configurations provided in the benchmark: all kernels in either device.

The **stencil benchmark** contains just one kernel, which is executed once. In this case the incompatibility and dependency matrices are 1x1 zero matrices. Also the feasibility matrix is a 1x2 matrix filled with ones. Profiling data is obtained for data sizes 256, 512, 1024, 2048, 4096, and 8192. These sizes correspond with the sides of the input matrices, such that base size 1024 corresponds with a matrix of real size 1024x1024 elements. We also consider all the corresponding 1/4, 2/4 and 3/4 fractions of each base size (For example 64, 128 and 192 are the fractions for the 256 data size). Then scheduling plans are obtained for data sizes 256, 512, 1024, 2048, 4096, and 8192. On each case the partition pattern used fragments the problems size in 16 slices. Therefore, the partition pattern goes from 0/16 of the total size to 16/16. Table 6.3 shows the scheduling plan obtained for each one of the data sizes considered. The reason for this process is twofold. Firstly, splitting in four parts seems to be the norm in similar state-of-the-art tools. The closest example is that of O'Boyle's static approach [55]. Secondly, we wanted to make sure that the partition sizes were multiples of the amount of partitions, and using interpolation up to 16 parts, we can also stress-test our algorithm.

Table 6.3: Stencil benchmark: Selected scheduling plans.

| DataSize | 256 | 512 | 1024 | 2048 | 4096 | 8192 |
|---|---|---|---|---|---|---|
| CPU partition | 16/16 | 16/16 | 16/16 | 4/16 | 4/16 | 4/16 |
| GPU partition | 0/16 | 0/16 | 0/16 | 12/16 | 12/16 | 12/16 |
| ACC partition | 0/16 | 0/16 | 0/16 | 0/16 | 0/16 | 0/16 |

Finally, each valid scheduling combination considered on previous configurations (data sizes from 256 to 8192 with a partition pattern from 0/16 to 16/16) is executed,

and all the performance results are compared with the performance of the selected solution.

The results for the stencil benchmark are shown in Figure 6.1, Figure 6.2 and Figure 6.3. Figure 6.1 shows the percentile of the performance of the selected solution considered within the whole performance distribution of all the valid scheduling combinations. Figure 6.2 compares the performance of the best and the worst configuration to the performance of the selected solution. The comparisons are done using absolute performances (best, worst and selected plan) as well as relative performance from the selected solution compared with the best and the worst ones (the bigger the percentage, the closer to the best solution it is). Figure 6.3 complements the previous figure. It shows how good is the proposed partition in a scale from the worst case, which represents 0%, to the best case, which represents 100%.

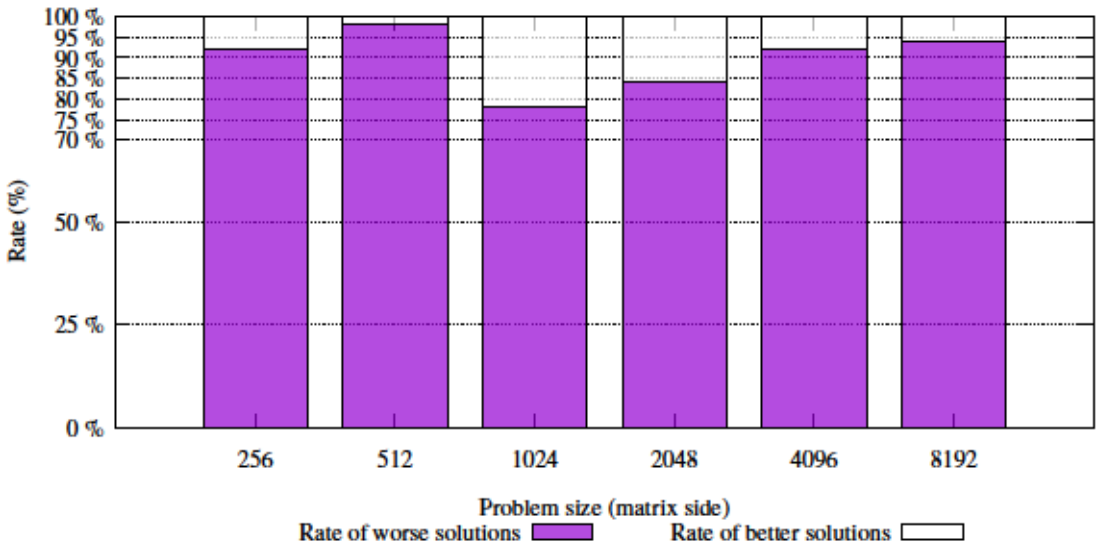


Figure 6.1: Stencil benchmark: Percentile of the obtained solution.

Several observations can be drawn from these results. Firstly, for small data sizes, the best option is to execute the whole kernel on the CPU. The reason behind this is that the data transfers to other devices cost outweighs other benefits. Besides, the problem size is not enough to take advantage of the level of parallelism that other devices offer. Secondly, for bigger data sizes, the GPU, and to some extent, the accelerator, exhibit a better performance ratio, but not enough to execute the whole kernel on the GPU. Therefore, the execution seems to settle down on a fixed execution ratio of 1/4 on the CPU and 3/4 on the GPU (at least for considered data sizes). Figure 6.1 shows that the performance of the selected solution is always very near to the best one. Besides, Figure 6.1 shows that there are very few solution (among all the solutions considered) with a better performance. In most cases the small discrepancy is due to the performance variability of the same scheduling over

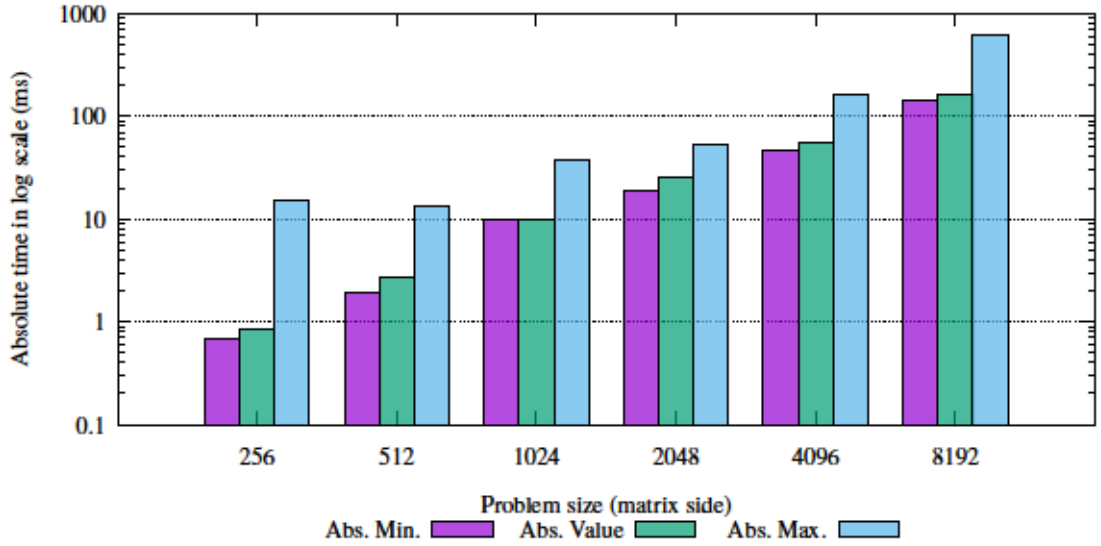


Figure 6.2: Stencil benchmark: Performance comparison.

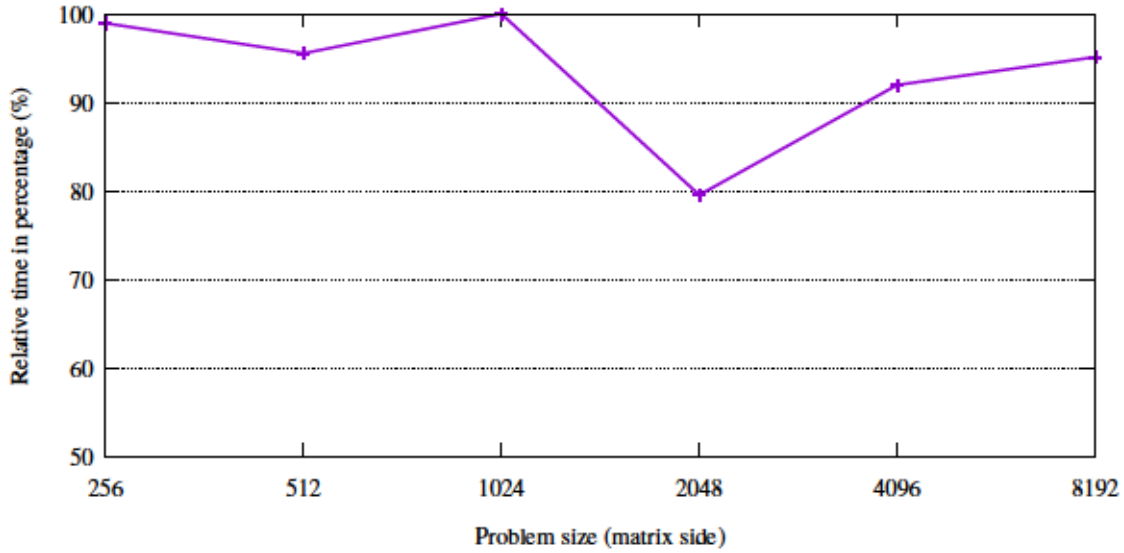


Figure 6.3: Stencil benchmark: Relative performance of the proposed partitioning.

several runs. However, 1024 and 2048 data sizes are slightly different because in those cases the difference between using only the CPU and a 1/4 CPU vs. 3/4 GPU usage is not as big. For example, for data size 1024 the percentile is lower but the performance is nearer to the best solution. This is because, in this case there are more solutions with a similar performance. A similar situation happens for data size 2048 but in this case performance ranking is worse due to the higher performance variability of the GPU compared to the CPU. Finally, in all tested

cases the accelerator has worse results than CPU and GPU. For that, the stencil model never takes into account the accelerator to process data. The reason behind this is that, while it is true that the accelerator may attain a better performance than the CPU, the cost of transferring data from the host to the accelerator negates most benefits. Furthermore, the GPU is better oriented towards data parallelism, and this means that, in the event that some kernel or partition can be delegated to another device, the GPU will still be a better choice than the accelerator. It is possible, however, that in cases of massive parallelism, where many kernels can be executed simultaneously, the accelerator will be used.

The **transitive closure benchmark** consists of three kernels, executed several times on a conditional loop. Each iteration depends on the result of the previous one. Consequently, we consider only the execution of one iteration. In this case, the incompatibility and dependency matrices are 3x3 zero matrices because all kernels are independent and required for each execution at the same time. Also, the feasibility matrix is a 3x2 matrix full of ones (all kernels can be executed on all the devices). Profiling data for the three kernels is obtained for data sizes 256, 512, 1024, 2048, 4096, and 8192. The comparison and the union kernels are executed only on CPU or GPU so no fractions are considered for the profiling data. For the multiplication kernel, 1/4, 2/4 and 3/4 fractions are also collected (For example 64, 128 and 192 are the fractions for the 256 data size). Scheduling plans are obtained for data sizes 128, 256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096, 6144, and 8192. For the multiplication kernel, the partition pattern used fragments the problems size in 16 slices. Therefore, the partition pattern goes from 0/16 of the total size to 16/16. Table 6.4 shows the scheduling plan obtained for each one of the data sizes considered.

Table 6.4: Transitive closure: Selected scheduling plans. X for CPU kernels and O for GPU kernels.

| DataSize | 128 | 256 | 384 | 512 | 768 | 1024 | 1536 | 2048 | 3072 | 4096 | 6144 | 8192 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Multi. Kernel | X | X | X | 3/4 X | X | X | X | X | X | X | X | X |
| Union Kernel | O | O | O | X | O | O | O | O | O | O | O | O |
| Comp. Kernel | X | X | X | O | O | O | O | O | O | O | O | O |

Finally, each valid scheduling combination considered in previous configurations (data sizes from 128 to 8192 with a partition pattern form 0/16 to 16/16 on the multiplication kernel) is executed, and all the performance results are compared with the performance of the selected solution.

Results for the transitive closure benchmark are shown in Figure 6.4, Figure 6.5 and Figure 6.6. Figure 6.4 shows the percentile of the performance of the selected solution considered within the whole performance distribution of all the valid scheduling combinations. Figure 6.5 compares the performance of the best and the worst configuration with the performance of the selected solution. The comparisons are

Table 6.5: S3D: Selected scheduling plans. X for CPU kernels and O for GPU kernels.

| Kernels | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Case 24 | X | O | O | O | O | O | O | O | O | O | O | O | O | O |
| Case 32 | X | O | O | O | O | O | O | O | O | O | O | O | O | O |
| Case 40 | X | O | O | O | O | O | O | O | O | O | O | O | O | O |
| Case 48 | X | O | O | O | O | O | O | O | O | O | O | O | O | O |

| Kernels | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Case 24 | O | O | O | O | O | X | O | O | O | O | O | O | X |
| Case 32 | O | O | O | O | O | O | O | O | O | O | O | O | X |
| Case 40 | O | O | O | O | O | O | O | O | O | O | O | O | X |
| Case 48 | O | O | O | O | O | O | O | O | X | X | X | X | X |

done by absolute performance (best, worst and selected schedule), as well as the relative performance of the selected solution compared to the best and the worst ones (the bigger the percentage, the closer to the best solution). Again, Figure 6.6 shows the performance of the proposed partitioning as a percentage, where 0% is the worst partition and 100% is the best.

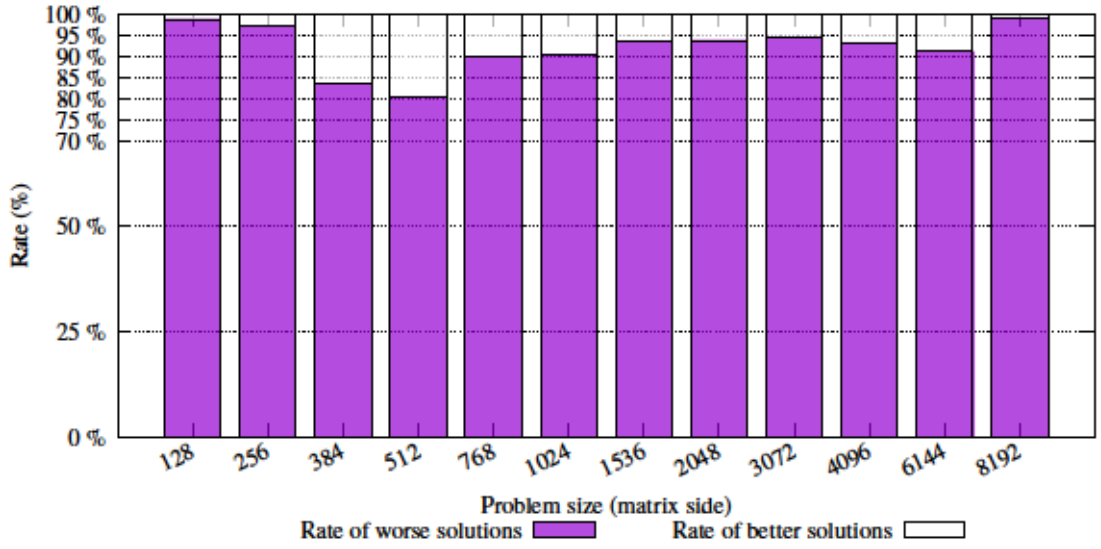


Figure 6.4: Transitive closure benchmark: Percentile of the obtained solution.

In this case observations from results are also relevant. Firstly, the multiplication kernel is almost always executed solely on the CPU. This is because the algorithm used is much more efficient on the CPU than on the GPU. Another kernel implementation would probably yield different results. As a consequence, the other two kernels are executed on the GPU in most cases. Only when the data size is small,

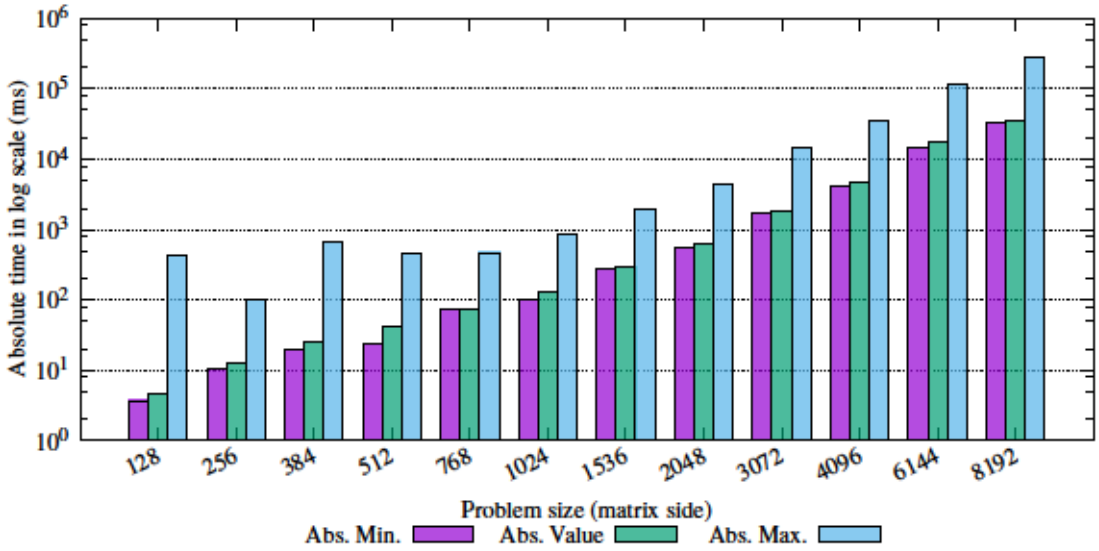


Figure 6.5: Transitive closure benchmark: Performance comparison.

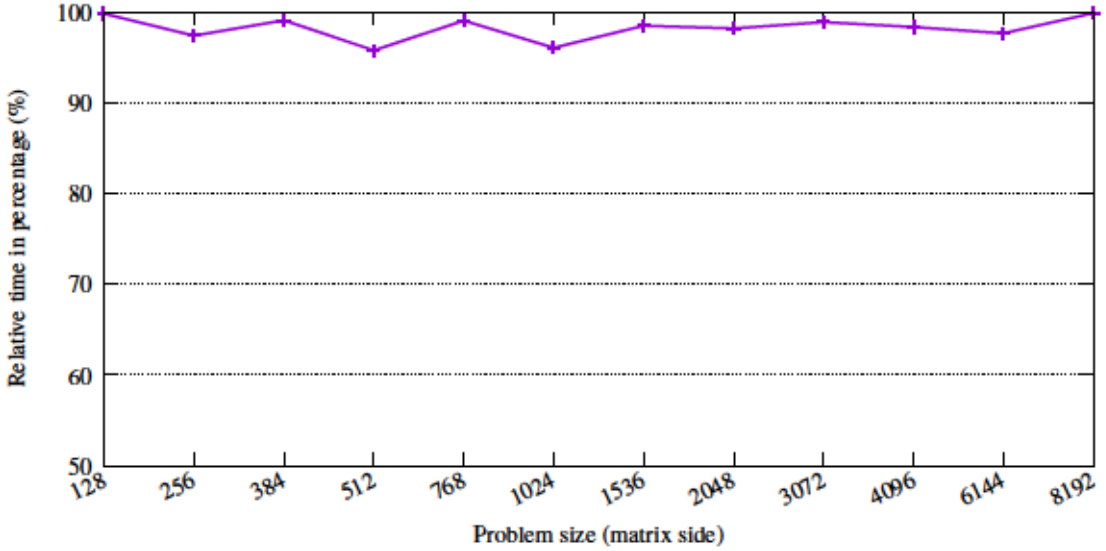


Figure 6.6: Transitive closure benchmark: Relative performance of the proposed partitioning.

one of these kernels is also executed on the CPU, as in those cases, avoiding the data transfers to the GPU is better than a greater parallelism degree. Figure 6.4 shows that there are very few solutions (amongst all the solutions considered) that exhibit a better performance. Besides, Figure 6.5 shows that the performance of the selected solution is always very near to the best one. In most cases, the small discrepancy is due to the performance variability of the same scheduling over several

executions. However 384 and 512 data sizes are slightly different because in those cases the difference between using mostly the CPU, or using the CPU only for the multiplication kernel is not as big. Therefore, there are more solutions with a similar performance and the corresponding percentile and execution time is a little worse.

The **S3D benchmark** consists of 27 kernels executed several times on a conditional loop. As with the transitive closure benchmark, we only consider the execution of a single iteration. The dependency matrix for S3D benchmark is the most complex of the three, since there are 27 kernels, separated in two distinct phases, and there are many dependencies between them. On the other hand, the incompatibility matrix, while bigger than the previous ones (27x27), is still a zero matrix because all kernels are compatible and required for each execution at the same time. The same happens with the feasibility matrix, given that all kernels can be executed on all the devices. Profiling data for all kernels is obtained for base data sizes 24, 32, 40 and 48. These sizes represent the basic size factor that is applied to all input and output parameters. All kernels are only executed on CPU or GPU, so no fractions are considered for the profiling data. Then, scheduling plans are obtained for the same data sizes, so that the scheduling algorithm will distribute the independent kernels between devices. Table 6.5 shows the scheduling plan obtained for each of the data sizes considered. It is worth to notice that most of the kernels are executed on the GPU and only a few of them are executed on the CPU.

Results show the comparison between the schedule obtained with the proposed algorithm, the execution of all the kernels on CPU and the execution of all the kernels on GPU (see Figure 6.7 and Figure 6.8). Results suggest executing everything in GPU is, probably, one of the options with better performance, while executing everything in CPU yields much worse execution time. However, sharing the computation tasks between both devices improves the results in most cases, even though sharing the computation involves many more data transfers than just using the GPU. It is remarkable that the GPU-only solution is a very good one, so the improvement is not very significant and can be lost due to variability of the executions and the error range of the algorithm. The obtained scheduling outperforms the GPU-only execution for cases with base data size 24, 32 and 48, although they are very close. However, in one case, with base data size of 40, the result obtained is worse than the GPU-only execution.

Figure 6.9 shows a summary that compares the results running the OpenCL implementation of a benchmark on a concrete device with the prediction model result. It includes all the benchmarks presented before, as well as an average of the results for all benchmarks. It shows that the model results are close or better that the best compared result.

As a rule of the thumb, the accelerator outperforms the other two devices in those cases where the execution units make use of vector instructions. Such examples are Intel's benchmarks, such as GEMM and Bitonic Sort. If the kernels don't make use of vector instructions, then either the GPU or the CPU yield better re-

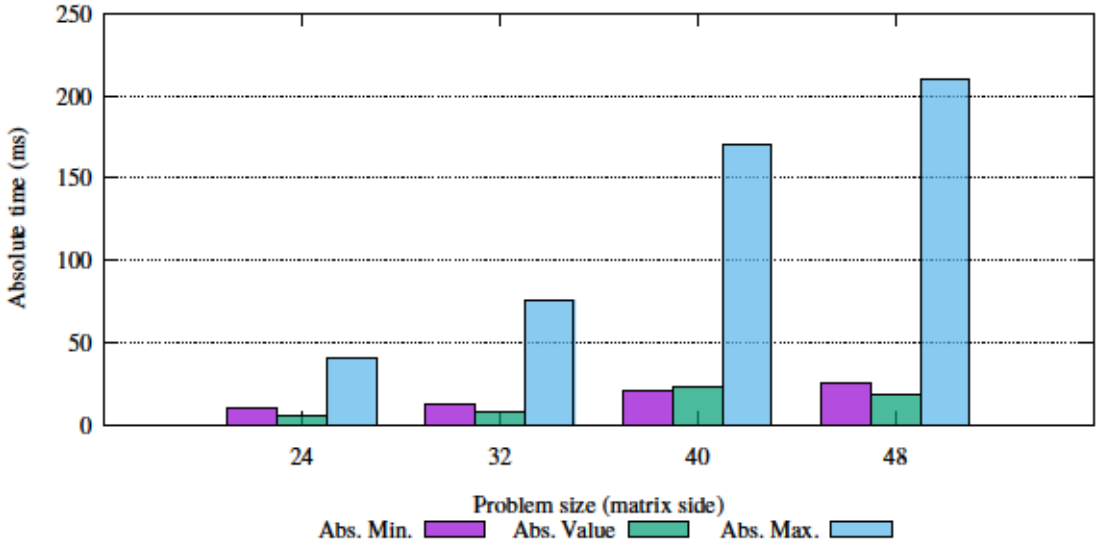


Figure 6.7: S3D benchmark: Performance comparison.

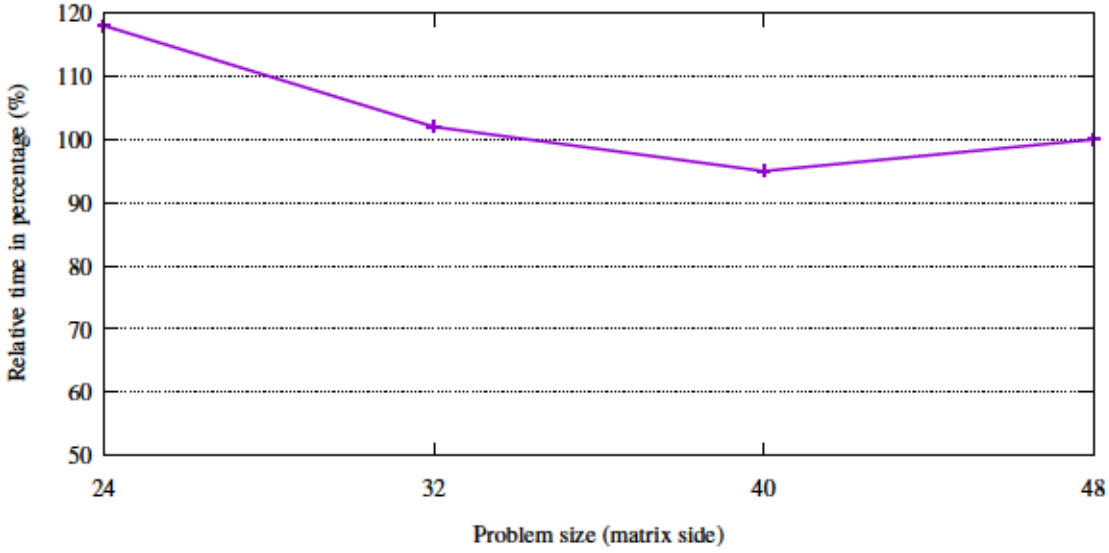


Figure 6.8: S3D benchmark: Relative performance of the proposed partitioning.

sults, depending on the kind of problem. GPU yields better results in purely SIMD problems, specially for bigger input data. For the rest of the cases, it is the CPU that gives the best performance. Our model manages to outperform these three separately because it considers execution and transfer times, as well as execution unit partitioning and multi-device scheduling.

One particular case where the model actually worsens the results is the GEMM benchmark for a base size of 256 elements. The reason for this is that, with small

data sizes, and because of the linear interpolation, it is possible that the model predicts a partition much too small to actually give some benefit. For this reason, a threshold is considered in small cases. This threshold cannot be too high, in order to avoid discarding too many possibilities. The GEMM benchmark was the single case where the prediction model fell outside the threshold, and still give a bad prediction.

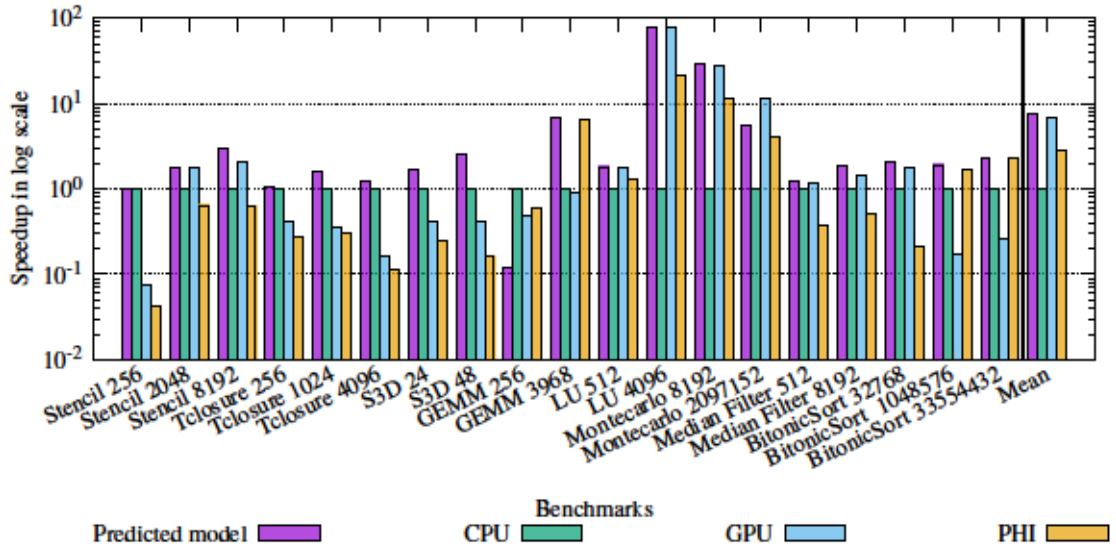


Figure 6.9: Summary of model prediction results.

# Chapter 7

# Conclusions and future work

In this Thesis, we have developed a framework to execute parallel code in heterogeneous parallel platforms with the objective of optimizing performance. Following is the current state of the original requirements for the goal of the Thesis:

1. **To describe the underlying hardware**. In Chapter 3 we have proposed a model that allows to describe the devices that make up an heterogeneous parallel platform and their capabilities, as well as how they are linked. Although the model is limited to four kinds of processors, it is designed so that extending it would be simple. Also, the description is detailed enough that external tools, like compilers, have information to optimize code for the more complex devices, such as FPGAs and DSPs.

2. **To describe the characteristics of the software.** In Chapter 4 we proposed the specifications for a set of necessary annotations needed to orchestrate the code. This annotations are standard, portable, and ensure the maintainability of the code. Along with this description come the techniques for automatically annotating kernels. Results so far show the feasibility of automatically detecting parallel pieces of code. Work is being done to automatically extract the information necessary to optimize the partitioning process, as well as detect higher level parallel patterns.

3. **A set of partitioning techniques** that work statically. These techniques, proposed in Chapter 5 are based on feedback obtained during run-time, and are the basis for the more advanced dynamic partitioning techniques. Results show that it is possible to operate in the top 5% of the best partitions.

## 7.1 Contributions

The following contributions have been reached during the development of the Thesis so far:

- **A hardware model** to represent the heterogeneous parallel platforms with sufficient level of detail. This is a model hierarchical model that represents devices of different families as components, or parts, of a unique heterogeneous platform. For each of this devices (e.g. CPU, GPU, FPGA or DSP), specific information is stored with regards to capabilities or limitations. This information may later be used to schedule code efficiently for execution in these devices.

- **A programming model**. This execution model is a host-accelerator model, not unlike that of OpenCL. It describes the interactions between the CPU and other external processors with regards to execution offloading and data movements.

- **A specification for annotating software**. This specification allows to identify potentially parallel pieces of code, referred to as *kernels*. It also allows to add semantic information regarding the orchestration of said kernels, such as input/output parameters, maximum expected memory usage, desired target devices or whether the kernel conforms to a specific parallel patten (e.g. map).

- **Automatic annotation techniques** that analyse a sequential code and annotates the potential kernels in an automatic fashion, taking care to exclude potentially parallel code that does not have a significant computational cost. These techniques also analyse the code for data dependencies and other data useful for optimizations.

- **Static scheduling techniques** used to efficiently execute the aforementioned kernels in the devices of an heterogeneous parallel platform in a parallel manner. For these techniques to work, the complete set of possible schedules is represented as a tree, where the root is an empty schedule, and the complete schedules (those where all kernels have been assigned to a device) are the leaf nodes. These schedules have an associated cost, and the best schedule is picked every time. This algorithm allows to partition a kernel between several devices based on linear interpolation of input sizes.

## 7.2 Dissemination

The contributions of this Thesis can be found in the following publications:

- **Journals**

  - R. Sotomayor, L. M. Sanchez, J.G. Blas, J. Fernandez, and J. D. Garcia. Automatic CPU/GPU generation of multi-versioned OpenCL kernels for C++ scientific applications. *International Journal of Parallel Programming*, 2016.

  - J. D. Garcia, R. Sotomayor, J. Fernandez, and L. M. Sanchez. Static partitioning and mapping techniques of kernel-based applications over modern heterogeneous architectures. *Simulation Modelling Practica and Theory*, 58, Part 1:79-94, 2015. Special Issue on Techniques and Applications for Sustainable Ultrascale Computing Systems.
  - L. M. Sanchez, J. Fernandez, R. Sotomayor, S. Escolar, and J. D. Garcia. A comparative Study and Evaluation of Parallel Programming Models for Shared-Memory Parallel Architectures. *New Generation Computing*, 31(3):139-161, 2013.

- **Conferences**

  - R. Sotomayor, J. D. Garcia. Application Partitioning and Mapping Techniques for Heterogeneous Parallel Platforms. In *Proceedings of the First PhD Symposium on Sustainable Ultrascale Computing Systems*, 2016.
  - M. Danelutto, J. D. Garcia, L. M. Sanchez, R. Sotomayor, and M. Torquati. Introducing parallelism by using REPARA C++11 attributes. In *24th Euromicro International Conference on Parallel, Distributed and Network Based Processing (PDP 2016)*, 2016.
  - R. Sotomayor, L. M. Sanchez, J. G Blas, A. Calderon, and J. Fernandez. AKI: Automatic Kernel Identification and annotation tool based on C++ attributes. In *IEEE TrustCom/BigDataSE/ISPA*, volume 3, pages 148-153, aug 2015.
  - R. Sotomayor, L. M. Sanchez, J.G. Blas, J. Fernandez, and J. D. Garcia. Automatic CPU/GPU generation of multi-versioned OpenCL kernels for C++ scientific applications. In *8th International Symposium on High-Level Parallel Programming and Applications*, 2015.
  - L. M. Sanchez, J. Fernandez, R. Sotomayor, and J. D. Garcia. A comparative evaluation of parallel programming models for shared memory architectures. In *Proceedings of the 2013 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA 2012, pages 363-370, Washington, DC, USA, 2012. IEEE Computer Society.

- **Upcoming**

  - D. del Rio, R. Sotomayor, L. M. Sanchez, J. Garcia, A. Calderon, and J. Fernandez. Assessing and discovering parallelism in C++ code for heterogeneous platforms. *Journal of Supercomputing*, 2016 (minor revision).

Additionally, the following products have been developed as part of the REPARA project:

- **Registered products**
  - AKI: Automatic Kernel Identification. Registration of the intellectual property. Code 09-RTPI-02819.5/2016.
  - SPT: Static Partitioning Tool. Registration of the intellectual property. Code 09-RTPI-02829.7/2016.
  - DPE: Dynamic Partitioning Engine. Registration of the intellectual property. Code 09-RTPI-02828.6/2016.
- **Technical reports**
  - D2.4: Algorithm constraining and selection techniques. Document detailing techniques for generating multiple versions of kernels and choosing between the best of them. ICT-609666-D2.4. http://repara-project.eu/?p=204
  - D3.1: Target platform description specification. Document detailing the hardware description language used to list the devices and capabilities of an heterogeneous parallel platform. ICT-609666-D3.1. http://repara-project.eu/?p=174
  - D3.3: Static partitioning tool. Set of techniques for statically partitioning code based on run-time feedback. ICT-609666-D3.3. http://repara-project.eu/?p=213
  - D3.5: Dynamic partitioning techniques. Document describing the set of techniques for dynamically partitioning code based on static metrics and run-time feedback in a flexible manner. ICT-609666-D3.5. http://repara-project.eu/?p=336
  - D6.2: Dynamic runtimes for heterogeneous platforms. Document describing the interactions and restrictions of the dynamic partitioning techniques with regards to parallel frameworks. ICT-609666-D6.2. http://repara-project.eu/?p=303
- **Grants**. Lastly, the development of the work so far has been possible thanks to the following **grants**:
  - Convocatoria PIF UC3M 01-1314 de Personal Investigador en Formación, PhD fully granted (4 years), 2013, Universidad Carlos III de Madrid.
  - Programa propio de investigación. Ayudas a la movilidad de investigadores en formación predoctoral. 3150, 2015, Universidad Carlos III de Madrid.

  The last grant has facilitated a research stay at Università di Pisa, under the supervision of Prof. Marco Danelutto from September 2015 to December 2015.

## 7.3 Future work

The work so far can be extended in several directions, based on the different contributions that have been listed in Section 7.1. This Section contains some current and potential ideas.

The first possibility is to adapt the static scheduling techniques to a dynamic environment, where a parallel framework may make use of the improved techniques and adapt the kernel mapping during runtime. This gives access to several possibilities, like dynamic optimization of stream computation.

A second line is to consider the static analysis of the code alongside the feedback obtained dynamically. So far, the static information may be used to make off-line decisions, like those made when automatically annotating the code. The current static, feedback-based algorithm, though adaptable, responds slowly to changes. For example, several calls to a kernel inside a utility library with different inputs may yield poor execution time for very disparate input sizes. By integrating static and dynamic information, it is possible to offset the latter and react with more precision to any such changes.

A third possible line is to improve data-partitioning techniques at the kernel level. One of the weaknesses of the partitioning algorithm so far is that it highly depends on the memory access pattern. A complex access pattern makes the process of partitioning the computation impossible, thus rendering the whole optimization useless. Currently, work is being done to detect the memory access patterns off-line with the *memcheck* tool of the Valgrind toolset to detect suitable memory patterns.

More interesting lines are based on the automatic annotation process, and the amount of information that can be obtained statically in order to optimize the orchestration of the code. For example, it is interesting to know beforehand if a given kernel or set of kernels fit the characteristics of a specific parallel pattern, such as a map, a farm or a pipeline.

# Bibliography


[1] Advantech. PCIe DSP Card. http://www.advantech.com/products/PCIe-DSP-Card/sub_Half-length_PCIe_Card1.aspx, Feb. 2014.

[2] O. E. Albayrak, I. Akturk, and O. Ozturk. Improving application behavior on heterogeneous manycore systems through kernel mapping. *Parallel Computing*, 39(12):867–878, 2013.

[3] M. Aldinucci, L. Anardu, M. Danelutto, M. Torquati, and P. Kilpatrick. Parallel patterns+ macro data flow for multi-core programming. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 27–36. IEEE, 2012.

[4] M. Aldinucci, M. Danelutto, M. Meneghin, M. Torquati, and P. Kilpatrick. Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed. In *Parallel Computing: From Multicores and GPU's to Petascale (PARCO)*, pages 273–280, 2009.

[5] M. Aldinucci, M. Meneghin, and M. Torquati. Efficient Smith-Waterman on multi-core with FastFlow. In *2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 195–199. IEEE, 2010.

[6] M. Aldinucci, G. P. Pezzi, M. Drocco, C. Spampinato, and M. Torquati. Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. *International Journal of High Performance Computing Applications*, 29(4):461–472, 2015.

[7] E. Alerstam, T. Svensson, and S. Andersson-Engels. Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration. *Journal of Biomedical Optics*, 13(6):060504–060504, 2008.

[8] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Proceedings of Programming Models for Massively Parallel Computers, 1993.*, pages 116–123. IEEE, 1993.

[9] Altera. OpenCL in FPGAs for GPU programmers. Technical report, Altera, jun 2014. https://www.altera.com/products/design-software/embedded-software-developers/opencl/documentation.html#White-Papers.

[10] Altera. Accelerating genomics research with OpenCL and FPGAs. Technical report, Altera, mar 2016. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[11] AMD. Amd-app-sdk-v3.0. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/, Feb. 2015.

[12] I. S. Association et al. IEEE Standard for Verilog Hardware Description Language. Design Automation Standards Committee, 2005. *IEEE Std 1364TM-2005*.

[13] A. Athavale, P. Ranadive, M. N. Babu, P. Pawar, S. Sah, V. Vaidya, and C. Rajguru. Automatic Sequential to Parallel Code Conversion The S2P Tool and Performance Analysis. *The GSTF Journal on Computing (JoC)*, 1(4), 2012.

[14] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par 2009 Parallel Processing*, pages 851–862. Springer, 2009.

[15] S. Baghdadi, A. Größlinger, and A. Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, Vienna, Austria, July 2010.

[16] Á. Baráth and Z. Porkoláb. Attribute-based checking of C++ move semantics. In *Proceedings of the 3rd Workshop on Software Quality Analysis, Monitoring, Improvement and Applications (SQAMIA 2014), Lovran, Croatia, September 19-22, 2014.*, pages 9–14, 2014.

[17] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In R. Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin Heidelberg, 2010.

[18] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[19] C. Bastoul. Extracting polyhedral representation from high level languages. Technical report, LRI, Paris-Sud University, 2008. http://icps.u-strasbg.fr/~bastoul/research/papers/clan.pdf.

[20] S. Benkner, S. Pllana, J. L. Träf, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov. PEPPHER: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41, 2011.

[21] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave. Coordinating GPU Threads for OpenMP 4.0 in LLVM. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 12–21, Piscataway, NJ, USA, 2014. IEEE Press.

[22] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[23] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The Journal of Political Economy*, pages 637–654, 1973.

[24] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995.

[25] O. A. R. Board. OpenMP compiler support. http://openmp.org/wp/openmp-compilers/, oct 2015.

[26] U. Bondhugula, V. Bandishti, A. Cohen, G. Potron, and N. Vasilache. Tiling and Optimizing Time-iterated Computations on Periodic Domains. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 39–50, New York, NY, USA, 2014. ACM.

[27] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. PLuTo: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ*, jun 2008.

[28] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. IEEE, 2010.

[29] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Trans. Archit. Code Optim.*, 12(1):6:1–6:24, Apr. 2015.

[30] B. S. Center. StarSs home page. https://www.bsc.es/publications/starss-programming-model-multicore-era, 2016.

[31] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[32] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[33] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11, Dec 2010.

[34] S. Chen. Lattice Boltzmann Method for Fluid Flows. In *AGU Fall Meeting Abstracts*, volume 1, page 04, 2001.

[35] Cray. OpenACC home page. http://www.openacc.org/node/1, 2015.

[36] D. Crockford. Introducing JSON. http://www.json.org/, Oct. 2013.

[37] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. P. Singh, and S. D. Brown. OpenCL for FPGAs: Prototyping a compiler. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[38] L. Dagum and R. Enon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[39] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, New York, NY, USA, 2010. ACM.

[40] M. Danelutto and M. Torquati. Loop parallelism: A new skeleton perspective on data parallel patterns. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 52–59. IEEE, 2014.

[41] M. Danelutto and M. Torquati. *Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*, chapter Structured Parallel Programming with "core" FastFlow, pages 29–75. Springer International Publishing, Cham, 2015.

[42] U. Dastgeer, J. Enmyren, and C. W. Kessler. Auto-tuning SkePU: A Multi-backend Skeleton Programming Framework for multi-GPU Systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 25–32, New York, NY, USA, 2011. ACM.

[43] U. Dastgeer, L. Li, and C. Kessler. The PEPPHER composition tool: Performance-aware dynamic composition of applications for GPU-based systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 711–720. IEEE, 2012.

[44] C. de la Lama, P. Toharia, J. Bosque, and O. Robles. Static multi-device load balancing for OpenCL. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 675–682, July 2012.

[45] J. Enmyren and C. W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[46] ezIX. Hardware Lister. `http://www.ezix.org/project/wiki/HardwareLiSter`, aug 2013.

[47] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[48] R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy. Columbus - reverse engineering tool and schema for C++. In *Proceedings of the International Conference on Software Maintenance*, pages 172–181, 2002.

[49] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[50] I. O. for Standarization. Information technology – programming languages, their environments and system software interfaces – ECMAScript language specification. Standard, ISO, jun 2011.

[51] GCC. GCC OpenACC experimental support. https://gcc.gnu.org/wiki/OpenACC, 2016.

[52] D. Göhringer and J. Tepelmann. An Interactive Tool Based on Polly for Detection and Parallelization of Loops. In *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '14, pages 1:1–1:6. ACM, 2014.

[53] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[54] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer. Libwater: Heterogeneous distributed computing made easy. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 161–172, New York, NY, USA, 2013. ACM.

[55] D. Grewe and M. F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 286–305, Berlin, Heidelberg, 2011. Springer-Verlag.

[56] D. Grewe, Z. Wang, and M. O'Boyle. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10, Feb 2013.

[57] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *In IEEE PACT*, pages 106–111. IEEE Computer Society Press, 1998.

[58] T. GROSSER, A. GROESSLINGER, and C. LENGAUER. Polly performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[59] T. K. Group. The OpenCL Extension Specification v2.0, Nov. 2013.

[60] E. R. Hawkes, R. Sankaran, J. C. Sutherland, and J. H. Chen. Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models. *Journal of Physics: Conference Series*, 16(1):65, 2005.

[61] Z. U. Huda, A. Jannesari, and F. Wolf. Using template matching to infer parallel design patterns. *ACM Trans. Archit. Code Optim.*, 11:64:1–64:21, Jan. 2015.

[62] T. Instruments. TI OpenCL v01.01.xx. http://downloads.ti.com/mctools/esd/docs/opencl/, 2015.

[63] Intel. Intel OpenCL code samples (2015). https://software.intel.com/en-us/intel-opencl-support/code-samples.

[64] Intel. Intel Hyper-Threading Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html, 2002.

[65] Intel. Cilk Plus. https://www.cilkplus.org/, sep 2010.

[66] Intel. Intel Cilk Plus tutorial: Array notation. http://www.cilkplus.org/tutorial-array-notation, Apr. 2013.

[67] Intel. Intel Language Extensions for Offload (LEO). https://software.intel.com/en-us/articles/xeon-phi-coprocessor-data-transfer-array-of-pointers-using-language-extensions-for-offload, 2014.

[68] Intel. The Parallel Universe. Technical report, Intel, 2014.

[69] Intel. Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/, 2015.

[70] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, Feb. 2012.

[71] H. Jin, G. Jost, J. Yan, E. Ayguade, M. Gonzalez, and X. Martorell. Automatic multilevel parallelization using OpenMP. *Sci. Program.*, 11(2):177–190, Apr. 2003.

[72] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. *SIGPLAN Not.*, 46(8):277–288, Feb. 2011.

[73] A. Koch and U. Golze. FPGA applications in education and research. *Technical University Braunschweig, Germany*, 1993.

[74] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 149–160, New York, NY, USA, 2013. ACM.

[75] B. J. LaMeres and C. Gauer. Dynamic reconfigurable computing architecture for aerospace applications. In *Aerospace conference, 2009 IEEE*, pages 1–6. IEEE, 2009.

[76] C. Y. Lee. An Algorithm for Path Connections and Its Applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, Sept 1961.

[77] S. Lee and J. S. Vetter. OpenARC: Open Accelerator Research Compiler for Directive-based, Efficient Heterogeneous Computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 115–120, New York, NY, USA, 2014. ACM.

[78] J. Levon and P. Elie. Oprofile: A system profiler for Linux. http://oprofile.sourceforge.net/news/, 2004.

[79] Z. Li, R. Atre, Z. Ul-Huda, A. Jannesari, and F. Wolf. DiscoPoP: A profiling tool to identify parallelization opportunities. In *Tools for High Performance Computing 2014*, chapter 3, pages 37–54. Springer International Publishing, Aug. 2015.

[80] G. License. gcov: Gnu coverage tool. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[81] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[82] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *2012 41st International Conference on Parallel Processing (ICPP)*, pages 48–57. IEEE, 2012.

[83] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet. Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1403–1410. ACM, 2009.

[84] O. Maitre, N. Lachiche, and P. Collet. Two ports of a full evolutionary algorithm onto GPGPU. In *Artificial Evolution*, pages 97–108. Springer, 2011.

[85] R. Martinkutė-Kaulienė, J. Stankevičienė, and S. Žinytė. Option pricing using Monte Carlo simulation. *Journal of Security & Sustainability Issues*, 2(4), 2013.

[86] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for parallel programming.* Pearson Education, 2004.

[87] C. Maxfield. *The design warrior's guide to FPGAs: devices, tools and flows.* Elsevier, 2004.

[88] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming.* Elsevier, 2012.

[89] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548, 2008.

[90] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergstrom. KernelGen - The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, pages 1011–1020, 2014.

[91] P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. In *Computers and Digital Techniques, IEE Proceedings-*, volume 152, pages 285–297. IET, 2005.

[92] Z. Navabi. *VHDL: Analysis and modeling of digital systems.* McGraw-Hill, Inc., 1997.

[93] M. Norman, J. Larkin, A. Vose, and K. Evans. A Case Study of CUDA FORTRAN and OpenACC for an Atmospheric Climate Kernel. *Journal of Computational Science*, 9:1 – 6, 2015.

[94] C. Nugteren and H. Corporaal. Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs. *ACM Trans. Archit. Code Optim.*, 11(4):35:1–35:25, Dec. 2014.

[95] C. Nugteren, R. Corvino, and H. Corporaal. Algorithmic species revisited: A program code classification based on array references. In *MuCoCoS '13: International Workshop on Multi-/Many-core Computing Systems*, 2013.

[96] C. Nugteren, P. Custers, and H. Corporaal. Automatic skeleton-based compilation through integration with an algorithm classification. In *APPT '13: Advanced Parallel Processing Technology*, 2013.

[97] NVidia. CUDA home page. http://www.nvidia.com/object/cuda_home_new.html, jun 2007.

[98] OpenCL. Open Computing Language. http://www.khronos.org/opencl, Nov. 2013.

[99] Par4All. Par4All Home page. http://www.par4all.org/, Apr. 2015.

[100] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. *Parallel Processing and Applied Mathematics: 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009. Revised Selected Papers, Part I*, chapter Fast In-Place Sorting with CUDA Based on Bitonic Sort, pages 403–410. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[101] PGI. PGI OpenACC compiler. https://www.pgroup.com/resources/accel.htm, 2016.

[102] PPCG. PPCG Home page. http://freecode.com/projects/ppcg, Apr. 2015.

[103] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[104] REPARA project. Target Platform Description Specification. http://repara-project.eu/?p=174, Feb. 2014.

[105] D. L. Rosenband and T. Rosenband. A design case study: CPU vs. GPGPU vs. FPGA. In *Formal Methods and Models for Co-Design, 2009. MEMOCODE'09. 7th IEEE/ACM International Conference on*, pages 69–72. IEEE, 2009.

[106] J. Russell and R. Cohn. *Processor Supplementary Capability*. Book on Demand, 2012.

[107] L. M. Sanchez, J. Fernandez, R. Sotomayor, S. Escolar, and J. D. Garcia. A Comparative Study and Evaluation of Parallel Programming Models for Shared-Memory Parallel Architectures. *New Generation Computing*, 31(3):139–161, 2013.

[108] M. Sandrieser, S. Benkner, and S. Pllana. Explicit platform descriptions for heterogeneous many-core architectures. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1292–1299. IEEE, 2011.

[109] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148, Nov 2011.

[110] J. Shen, A. L. Varbanescu, and H. Sips. Look before you leap: Using the right hardware resources to accelerate applications. In *Proceedings of IEEE 16th International Conference on High Performance Computing and Communications (HPCC 2014)*, page 9, August 2014.

[111] J. Shen, A. L. Varbanescu, P. Zou, Y. Lu, and H. J. Sips. Improving performance by matching imbalanced workloads with heterogeneous platforms. In *2014 International Conference on Supercomputing, ICS'14, Muenchen, Germany, June 10-13, 2014*, pages 241–250, 2014.

[112] A. Sloss, D. Symes, and C. Wright. *ARM System Developer's Guide: Designing and Optimizing System Software*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[113] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[114] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[115] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30(3):202–210, 2005.

[116] O. Team. Portable Hardware Locality. https://www.open-mpi.org/projects/hwloc/, dec 2015.

[117] The Khronos Group. The OpenCL Specification v2.0. http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, Nov. 2013.

[118] M. e. a. Torquati. An innovative compilation tool-chain for embedded multi-core architectures. In *Embedded World Conference (February 2012)*, 2012.

[119] G. Tournavitis and B. Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 377–388, New York, NY, USA, 2010. ACM.

[120] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[121] C. Trapnell and M. C. Schatz. Optimizing data intensive GPGPU computations for DNA sequence alignment. *Parallel computing*, 35(8):429–440, 2009.

[122] U. o. T. University of Pisa. FastFlow home page. http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:about, 2009.

[123] H. Vandierendonck, S. Rul, and K. De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *Parallel Architectures and Compilation Techniques, 19th International Conference, Proceedings*, pages 389–400. Association for Computing Machinery (ACM), 2010.

[124] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson. Comparing hardware accelerators in scientific applications: A case study. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):58–68, 2011.

[125] P. Wesseling. *Principles of computational fluid dynamics*, volume 29. Springer Science & Business Media, 2009.

[126] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC: First Experiences with Real-world Applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

[127] R. Wiśniewski. *Synthesis of compositional microprogram control units for programmable devices*. University of Zielona Góra, 2009.

[128] Xilinx. Ultra High Definition TVs with FPGAs. http://www.xilinx.com/applications/consumer-electronics/ultra-high-definition-televisions.html.

# Appendices

# Appendix A

# HPP-DL

## A.1 Components

This section shows how to describe the different components according to the HPP-DL specification.

### A.1.1 Platform

This section shows how to describe the platform component according to the HPP-DL specification.

**Attributes** Table A.1 shows the specific attributes for the platform schema.

**Example** Listing A.1 shows a sample of platform specification in HPP-DL.

Listing A.1: Platform example

```
{
        "class": "platform",
        "id": "platform:0",
        "description": "REPARA Reference System. X9DRG-QF (To be filled by O.E.M.)",
        "model": "X9DRG-QF",
        "vendor": "Supermicro Inc.",
        "numa_nodes": 2,
        "processors": 2,
        "cores": 12,
        "pu_num": 24,
        "global_mem_size": "64 GiB",
        "capabilities": []
}
```

Table A.1: Platform attributes.

| Name | Type | Description |
|---|---|---|
| class | string | Component type. The value for this kind of object is `hpp`. |
| id | string | Component ID: `platform:0` |
| description | string | Component description. |
| vendor | string | Platform vendor. |
| model | string | Platform model. |
| numa_nodes | number | Number of NUMA nodes in the platform. |
| processors | number | Number of processors counting all the nodes. |
| cores | number | Number of cores available between all the processors. |
| pu_num | number | Number of processing units of the platform |
| global_mem_size | magnitude | Memory size capacity, described using IEC binary suffixes. |
| capabilities | string[] | List of capabilities. Empty for this component |

Listing A.2: Memory example

```
{
        "class": "memory",
        "id": "platform:0.memory:0",
        "description": "System Memory ",
        "model": "",
        "vendor": "",
        "size": "16 GiB",
        "num_banks":2,
        "capabilities":[]
}
```

### A.1.2 Memory

This section shows how to describe the memory component according to the HPP-DL specification.

**Attributes** Table A.2 shows the specific attributes for memory schema.

**Examples** Listing A.2 shows a sample of memory specification in HPP-DL.

Table A.2: Memory attributes.

| Name | Type | Description |
|---|---|---|
| class | string | Component type. The value is `core`. |
| id | string | Component ID. |
| description | string | Component description. |
| vendor | string | Platform vendor. In this case, it takes the default value "" |
| model | string | Platform model. In this case, it takes the default value "" |
| size | magnitude | Represents the size of all memory banks associate to this memory. Memory size capacity using IEC binary prefixes. |
| num_banks | number | Number of associated memory banks that form the memory component. |
| capabilities | string[] | List of capabilities. In this case, it takes the default value [] |

Listing A.3: Memory bank example

```
{
        "class":"bank",
        "id": "platform:0.memory:0.bank:0",
        "description": "DIMM DDR3 1333 MHz (0,8 ns)",
        "model": "M393B1K70DH0-CK0",
        "vendor": "Samsung",
        "serial": "1484586A",
        "slot": "P1-DIMMA1",
        "size": "8GiB",
        "width": "64 bits",
        "clock": "1333MHz",
        "latency":"0.8ns",
        "generation":"DDR3",
        "family":"DIMM",
        "architecture":"SDRAM",
        "capabilities":[]
}
```

### A.1.3 Memory banks

This section shows how to describe the memory bank component according to the HPP-DL specification.

**Attributes** Table A.3 shows the specific attributes for memory schema.

**Examples** Listing A.3 shows a sample of memory bank specification in HPP-DL.

Table A.3: Memory bank attributes.

| Name | Type | Description |
|---|---|---|
| class | string | Component type. The value is `core`. |
| id | string | Component ID. |
| description | string | Component description |
| vendor | string | Platform vendor. |
| model | string | Platform model. |
| slot | string | Physical slot on the board (platform, FPGA, etc.) |
| width | number | Word size on bits. |
| clock | magnitude | Speed on MHz. |
| latency | magnitude | Access latency in nanoseconds. |
| generation | string | Generation to which the memory bank belongs (i.e. DDR3, DDR5, etc.). |
| family | string | Specific memory type (i.e. DIMM, SIMM, etc.) |
| architecture | string | Architecture type (e.g. DRAM, SDRAM) |
| capabilities | string[] | List of capabilities. In this case, it takes the default value [] |

### A.1.4 Processor

This section shows how to describe the processor component according to the HPP-DL specification.

**Attributes** Table A.4 shows the specific attributes of the processor component.

`num_pu` attribute

`numa_group` attribute System designers use non-uniform memory access (NUMA) to increase processor speed without increasing the load on the processor bus. The architecture is non-uniform because each processor is close to some parts of memory and farther from other parts of memory. The processor quickly gains access to the memory it is close to, while it can take longer to gain access to memory that is farther away. In a NUMA system, processors are arranged in smaller systems called `nodes`. Each `node` has its own processors and memory, and is connected to the larger system through a cache-coherent interconnecting bus. A `node` is identifying with an ID.

**Examples** Listing A.4 shows a sample of processor specification in HPP-DL.

Table A.4: CPU Processor attributes.

| Name | Type | Description |
|---|---|---|
| class | string | Component type. The value for this kind of object is `processor`. |
| id | string | Component ID. |
| description | string | Component description. |
| model | string | Specific ID model. |
| vendor | string | Vendor of product. |
| cores | number | Number of cores of a processor. |
| pu_num | number | Number of PUs of a processor. |
| numa_group | number | NUMA group ID where the processor is included. |
| capabilities | string[] | List of capabilities. |

Listing A.4: processor example

```
{
        "class": "processor",
        "id": "platform:0.processor:0",
        "description": "Intel (R) Xeon(R) CPU E5-2620 0 @ 2.00GHz",
        "model": "E5-2620",
        "vendor": "Intel Corp.",
        "cores": 6,
        "pu_num":12,
        "numa_group":1, /* NUMA group ID where the processor is included */
        "capabilities": []
}
```

### A.1.5 Core

This section shows how to describe the core component according to the HPP-DL specification.

**Attributes** Table A.5 shows the specific attributes. Section A.4 (p. 160) summarizes the capability flags for different core components.

**Examples**

### A.1.6 CPU core

Listing A.5 shows a sample of CPU core specification in HPP-DL.

Table A.5: Core specific attributes.

| Name | Type | Description |
|---|---|---|
| class | string | Component type. The value is `core`. |
| id | string | Component ID: `<previous path>.core:<unsigned int>`. |
| description | string | Component description |
| model | string | Specific model. |
| vendor | string | Vendor of product. |
| clock | magnitude | Clock of the device. It should be expressed as GHz or MHz. |
| width | number | Word size expressed in bits used by the core. |
| GFLOPS | number | Floating-point Operations Per Second. |
| GMACS | number | Giga Multiply-Accumulate Operations per Second (aka Giga Multiply-Accumulates per Second). |
| architecture | string | Hardware architecture of the core. |
| pu_num | number | Number of PUs of the core. |
| pu_list | number[] | Unique IDs of PUs that run in the core. |
| capabilities | string[] | List of capabilities. |

Listing A.5: CPU core example

```
{
        "class": "core",
        "id": "platform:0.processor:0.core:0",
        "description": "Intel (R) Xeon(R) CPU E5-2620 0 @ 2.00GHz",
        "model": "E5-2620",
        "vendor": "Intel Corp.",
        "clock":"2.00GHz",
        "architecture":"x86_64",
        "GFLOPS": 16.8,
        "GMACS":0,
        "width": 64,
        "pu_num":2,
        "pu_list": [0,10],
        "capabilities": ["x86-64", "fpu", "fpu_exception", "wp", "vme", "de", "pse"
            , "tsc", "msr", "pae", "mce", "cx8", "apic", "sep", ...]
}
```

### A.1.7 DSP core

Listing A.6 shows a sample of DSP core specification in HPP-DL.

Listing A.6: DSP core example

```
{
        "class":"core",
        "id":"platform:0.processor:0.core:3",
        "description":"TMS320C6678",
        "model":"TMS320C6678",
        "vendor":"Texas Instruments",
        "clock":"1.25 GHz",
        "architecture":"TMS320C66x",
        "GFLOPS": 160,
        "GMACS":320",
        "width":0,
        "pu_num":1,
        "pu_list": [0],
        "capabilities":[...]
}
```

Listing A.7: FPGA core example

```
{
        "class": "core",
        "id": "platform:0.fpga:0.core:0",
        "description": "XC7VX485T-2FFG1761CES",
        "model": "XC7VX485T-2FFG1761CES",
        "vendor": "Xilinx Inc.",
        "clock": "200MHz",
        "architecture": "Xilinx",
        "GFLOPS": 0,
        "GMACS":0,
        "width":0,
        "pu_num":1,
        "pu_list": [0],
        "capabilities":[...]
}
```

### A.1.8 FPGA core

Listing A.7 shows a sample of FPGA core specification in HPP-DL.

### A.1.9 Cache

This section shows how to describe the cache component according to the HPP-DL specification.

**Attributes** Table A.6 shows the specific attributes for cache schema. Section A.4 (p. 162) summarizes the capability flags for cache component.

**Examples** Listing A.8 shows a sample of cache memory specification in HPP-DL.

Table A.6: Cache attributes.

| Name | Type | Description |
|---|---|---|
| class | string | Component type. The value for this kind of object is `cache`. |
| id | string | Component ID: `<path>.cache:<unsigned int>` |
| description | string | Component description. |
| model | string | Specific model. In this case, it takes the value `"'L' + level"` |
| vendor | string | Vendor of product. In this case, it takes the value `""` |
| size | magnitude | Memory size capacity. It is represented using IEC binary prefixes. |
| level | number | Cache level (from 1 to N). |
| ways_of_associativity | number | The number of subsets in the related cache memory. |
| coherency_line_size | number | Access size for operations to/from memory. |
| number_of_sets | number | The number of blocks per set. Depends on cache layout (e.g. direct mapped, set-associative, or fully associative). |
| physical_line_partition | number | Number of physical lines of a partition. |
| capabilities | string[] | List of capabilities. `"data"`, `"instruction"` or `"unified"` are exclusive |

### A.1.10 General-Purpose Computing on Graphics Processing Units (GPGPU)

This section shows how to describe the GPGPU component according to the HPP-DL specification.

**Attributes** The attributes defined for a GPGPU are obtained from the OpenCL 2.0 specification [117]. The following tables include all attributes used for describing GPGPU and their corresponding value on OpenCL. The attributes are classified by categories. Some attributes are not supported by previous OpenCL device versions, and include a equivalent OpenCL if it is possible. These attributes include a version

Listing A.8: Cache example

```
{
        "class":"cache",
        "id": "platform:0.processor:1.cache:0",
        "description": "L1 cache",
        "model": "",
        "vendor": "",
        "level": 1,
        "slot": "L1 Cache",
        "size": "32KiB",
        "ways_of_associativity": 8,
        "coherency_line_size":64,
        "number_of_sets":64,
        "physical_line_partition": 1,
        "capabilities": ["internal", "write-through", "data"]
}
```

number (v2.0, v1.2, or v1.1).

Table A.7 shows the common attributes for a component. Table A.8 shows the attributes that define the version of a GPGPU. Table A.9 describes the floating-point precision attributes for GPGPUs. Tables A.10, A.11 and A.12 show the general attributes for describing a GPGPU.

Table A.13 shows the memory attributes of a GPGPU component. Table A.14 shows the attributes for image management and their corresponding value in OpenCL. Table A.15 includes the queue and pipe attributes uses for the communication with the GPGPU and kernel management. The queue values describe the command-queue properties supported by the device. This attributes obtain their value form the equivalent value in OpenCL.

Table A.16 shows the pipe characteristics for GPGPUs. A pipe is a memory object that stores data organized as a FIFO. If pipe is not supported by the GPGPU (above OpenCL v2.0), all the attributes take the value 0.

Tables A.17 and A.18 represent the preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. Some attributes could be 0 if a concrete data type vector is not supported.

For all Tables, value `CL_TRUE` is `"1"` and `CL_FALSE` is `"0"`.

Table A.7: GPGPU common attributes.

| Name | Description |
|---|---|
| class (string)<br>- | Component type. The value for this kind of object is `gpgpu`. |
| description (string)<br>CL_DEVICE_NAME | Device name string. |
| vendor (string)<br>CL_DEVICE_VENDOR | Vendor name string. |
| model (string)<br>CL_DEVICE_NAME | Device name string. |
| capabilities (string[])<br>CL_DEVICE_EXTENSIONS | Space-separated list<br>of extension names<br>do not contain any spaces<br>Currently can include one<br>or more of the OpenCL<br>approved extension names. |

Table A.8: GPGPU version attributes.

| Name | Description |
|---|---|
| driver_version (string)<br>CL_DRIVER_VERSION | OpenCL software driver version<br>string in the form<br>`major_number.minor_number`. |
| device_version (string)<br>CL_DEVICE_VERSION | OpenCL version string.<br>Returns the OpenCL version<br>supported by the device. |

Table A.9: Floating-point precision attributes for GPGPUs.

| Name | Description |
|---|---|
| single_fp_config (number)<br>CL_DEVICE_SINGLE_FP_CONFIG | Describes single precision<br>floating-point capability. |
| half_fp_config (number)<br>CL_DEVICE_HALF_FP_CONFIG | Describes half precision<br>floating-point capability. |
| double_fp_config (number)<br>CL_DEVICE_DOUBLE_FP_CONFIG | Describes double precision<br>floating-point capability. |

Table A.10: GPGPU general compute attributes 1.

| Name | Description |
|---|---|
| address_bits (number)<br>CL_DEVICE_ADDRESS_BITS | Default compute device address.<br>Currently 32 or 64 bits |
| built_in_kernels (string)<br>CL_DEVICE_BUILT_IN_KERNELS | List of built-in kernels.<br>supported by the device.<br>An empty string is returned if no<br>built-in kernels are supported. |
| preferred_interop_user_sync (number)<br>CL_DEVICE_PREFERRED_INTEROP_USER_SYNC | `CL_TRUE` if the user<br>is responsible for synchronization<br>of shared memory objects between OpenCL and<br>other APIs such as DirectX, CL_FALSE<br>otherwise. |
| endian_little (number)<br>CL_DEVICE_ENDIAN_LITTLE | Is `CL_TRUE` if the OpenCL<br>device is a little endian device and<br>`CL_FALSE` otherwise. |
| error_correction_support (number)<br>CL_DEVICE_ERROR_CORRECTION_SUPPORT | Is `CL_TRUE` if the device<br>implements error correction for all accesses<br>to compute device memory (global and constant).<br>Is `CL_FALSE` if the device does not<br>implement such error correction. |
| execution_capabilities (number)<br>CL_DEVICE_EXECUTION_CAPABILITIES | Describes the execution<br>capabilities of the device. |

Table A.11: GPGPU general compute attributes 2.

| Name | Description |
|---|---|
| profiling_timer_resolution (number)<br>CL_DEVICE_PROFILING_TIMER_RESOLUTION | Describes the resolution of the device timer. It is measured in nanoseconds. |
| max_clock_frequency (number)<br>CL_DEVICE_MAX_CLOCK_FREQUENCY | Maximum configured clock frequency of the device in MHz. |
| max_compute_units (number)<br>CL_DEVICE_MAX_COMPUTE_UNITS | The number of parallel compute units on the OpenCL device. A work-group runs on a single compute unit. The minimum value is 1. |
| max_work_group_size (number)<br>CL_DEVICE_MAX_WORK_GROUP_SIZE | Maximum number of work-items in a work-group executing a kernel on a single compute unit, using the data parallel execution model. |
| max_work_item_dimensions (number)<br>CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS | Maximum dimensions that specify the global and local work-item IDs used by the data parallel execution model. |

Table A.12: GPGPU general compute attributes 3.

| Name | Description |
|---|---|
| max_work_item_sizes (number)<br>CL_DEVICE_MAX_WORK_ITEM_SIZES | Maximum number of work-items that can be specified in each dimension of the work-group. |
| max_constant_args (number)<br>CL_DEVICE_MAX_CONSTANT_ARGS | Max number of arguments declared with the `__constant` qualifier in a kernel. |
| max_samplers (number)<br>CL_DEVICE_MAX_SAMPLERS | Maximum number of samplers that can be used in a kernel. |
| max_parameter_size (number)<br>CL_DEVICE_MAX_PARAMETER_SIZE | Max size in bytes of all arguments that can be passed to a kernel. |
| GFLOPS (number)<br>- | Floating-point Operations Per Second. |

Table A.13: GPGPU memory attributes.

| Name | Type | Corresponding OpenCL value | Description |
|---|---|---|---|
| host_unified_memory | number | CL_DEVICE_HOST_UNIFIED_MEMORY (v1.2) | Is `CL_TRUE` if the device and the host have a unified memory subsystem and is `CL_FALSE` otherwise. |
| global_mem_cache_size | number | CL_DEVICE_GLOBAL_MEM_CACHE_SIZE | Size of global memory cache in bytes. |
| global_mem_cache_type | number | CL_DEVICE_GLOBAL_MEM_CACHE_TYPE | Type of global memory cache supported. |
| global_mem_cacheline_size | number | CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE | Size of global memory cache line in bytes. |
| global_mem_size | number | CL_DEVICE_GLOBAL_MEM_SIZE | Size of global device memory in bytes. |
| local_mem_size | number | CL_DEVICE_LOCAL_MEM_SIZE | Size of local memory region in bytes. |
| local_mem_type | number | CL_DEVICE_LOCAL_MEM_TYPE | Type of local memory supported. |
| max_constant_buffer_size | number | CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE | Maximum configured clock frequency of the device in MHz. |
| max_mem_alloc_size | number | CL_DEVICE_MAX_MEM_ALLOC_SIZE | Max size of memory object allocation in bytes. |
| mem_base_addr_align | number | CL_DEVICE_MEM_BASE_ADDR_ALIGN | The minimum value is the size (in bits) of the largest OpenCL built-in data type supported by the device (long16 in FULL profile, long16 or int16 in EMBEDDED profile) for devices that are not of type `CL_DEVICE_TYPE_CUSTOM`. |
| svm_capabilities | number | CL_DEVICE_SVM_CAPABILITIES (v2.0) | Describes the various shared virtual memory (a.k.a. SVM) memory allocation types the device supports. Coarse-grain SVM allocations are required to be supported by all OpenCL 2.0 devices. |

Table A.14: GPGPU image support.

| Name | Description |
|---|---|
| image_support (number) Is CL_TRUE if images are supported by CL_DEVICE_IMAGE_SUPPORT | the OpenCL device and CL_FALSE otherwise. |
| max_write_image_args (number[])<br><br>CL_DEVICE_MAX_WRITE_IMAGE_ARGS | Max number of image objects arguments of a<br>kernel declared with the write_only or read_only qualifier. |
| max_read_image_args (number)<br><br>CL_DEVICE_MAX_READ_IMAGE_ARGS | Max number of image objects arguments of a<br>kernel declared with the read_only qualifier. |
| image2d_max_height (number)<br>CL_DEVICE_IMAGE2D_MAX_HEIGHT | Max height of 2D image in pixels. |
| image2d_max_width (number)<br>CL_DEVICE_IMAGE2D_MAX_WIDTH | Max width of 2D image or 1D image not created from a buffer object in pixels. |
| image3d_max_depth (number)<br>CL_DEVICE_IMAGE3D_MAX_DEPTH | Max depth of 3D image in pixels. |
| image3d_max_height (number)<br>CL_DEVICE_IMAGE3D_MAX_HEIGHT | Max height of 3D image in pixels. |
| image3d_max_width (number)<br>CL_DEVICE_IMAGE3D_MAX_WIDTH | Max width of 3D image in pixels. |
| image_pitch_alignment (number)<br>CL_DEVICE_IMAGE_PITCH_ALIGNMENT | The row pitch alignment size in pixels for 2D images created from a buffer.<br>The value returned must be a power of 2. |
| image_base_address_alignment (number)<br>CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT | This query should be used when a 2D image is created from a buffer. The value returned must be a power of 2. |

Table A.15: GPGPU queue support.

| **Name** | **Description** |
|---|---|
| max_on_device_queues (number)<br>CL_DEVICE_MAX_ON_DEVICE_QUEUES (v2.0) | The maximum number of device queues that can be created per context. The minimum value is 1. |
| queue_on_host_properties (number)<br><br>CL_DEVICE_QUEUE_ON_HOST_PROPERTIES (v2.0)/<br>CL_DEVICE_QUEUE_PROPERTIES (v1.2) | Describes the on host command-queue properties<br>supported by the device. |
| queue_on_device_max_size (number)<br>CL_DEVICE_QUEUE_ON_DEVICE_MAX_SIZE (v2.0) | The max. size of the device queue in bytes. The minimum value is 256 KB for the full profile<br>and 64 KB for the embedded profile. |
| queue_on_device_preferred_size (number)<br><br>CL_DEVICE_QUEUE_ON_DEVICE_PREFERRED_SIZE (v2.0) | The size of the device queue in bytes preferred<br>by the implementation. Applications should use this size for the device queue to ensure<br>good performance. The minimum value is 16 KB. |

Table A.16: GPGPU pipe support.

| Name | Description |
|---|---|
| max_pipe_args (number)<br>CL_DEVICE_MAX_PIPE_ARGS (v2.0) | The maximum number of pipe objects that can be passed as arguments to a kernel. The minimum value is 16. |
| pipe_max_active_reservations (number)<br><br>CL_DEVICE_PIPE_MAX_ACTIVE_RESERVATIONS (v2.0) | The maximum number of reservations that can<br>be active for a pipe per work-item in a kernel. A work-group reservation is counted as one reservation<br>per work-item. The minimum value is 1. |
| pipe_max_packet_size (number)<br>CL_DEVICE_PIPE_MAX_PACKET_SIZE (v2.0) | The maximum size of pipe packet in bytes. The minimum value is 1024 bytes. |

Table A.17: GPGPU vector attributes 1.

| Name | Description |
|---|---|
| native_vector_width_char (number)<br>CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR | Returns the native ISA vector width.<br>The vector width is defined as the number of scalar elements that can be stored in the vector. |
| native_vector_width_double (number)<br>CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE | Equivalent to `native_vector_width_char` |
| native_vector_width_float (number)<br>CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT | Equivalent to `native_vector_width_char` |
| native_vector_width_half (number)<br>CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF | Equivalent to<br>`native_vector_width_char` |
| native_vector_width_int (number)<br>CL_DEVICE_NATIVE_VECTOR_WIDTH_INT | Equivalent to<br>`native_vector_width_char` |
| native_vector_width_long (number)<br>CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG | Equivalent to<br>`native_vector_width_char` |

Table A.18: GPGPU vector attributes 2.

| Name | Description |
|---|---|
| preferred_vector_width_char (number)<br>CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR | Preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar<br>elements that can be stored in the vector. |
| preferred_vector_width_double (number)<br>CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE | Equivalent to<br>`preferred_vector_width_char` |
| preferred_vector_width_float (number)<br>CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT | Equivalent to<br>`preferred_vector_width_char` |
| preferred_vector_width_half (number)<br>CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF | Equivalent to<br>`preferred_vector_width_char` |
| preferred_vector_width_int (number)<br>CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT | Equivalent to<br>`preferred_vector_width_char` |
| preferred_vector_width_long (number)<br>CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG | Equivalent to<br>`preferred_vector_width_char` |

**Examples** Listings A.9, A.10 and A.11 show a sample of GPGPU board specification in HPP-DL.

Listing A.9: GPGPU example 1

```
{
        /* Common attributes */
        "class": "gpgpu",
        "id": "platform:0.gpgpu:0,
        "description": "geforce gtx titan",
        "model": "GK110",
        "vendor": "nvidia corporation",
        /* General compute attributes */
        "address_bits": 32,
        "error_correction_support" :1,
        "built_in_kernels":"",
        "preferred_interop_user_sync":0,
        "endian_little": 1,
        "error_correction_support":0,
        "execution_capabilities":0,
        "profiling_timer_resolution": 1000,
        "max_clock_frequency": 1000,
        "max_compute_units": 14,
        "max_work_group_size": 1024,
        "max_work_item_dimensions": 3,
        "max_work_item_sizes": [1024,1024,64],
        "max_constant_args": 9,
        "max_parameter_size": 4352,
        "max_samplers": 32,

        /* Capabilities */
        "capabilities": ["cl_khr_byte_addressable_store", "cl_khr_d3d10_sharing", "
            cl_khr_fp64", "cl_khr_gl_sharing", "cl_khr_global_int32_base_atomics",
            "cl_khr_global_int32_extended_atomics", "cl_khr_icd", "
            cl_khr_local_int32_base_atomics", "cl_khr_local_int32_extended_atomics"
            , "cl_nv_compiler_options", "cl_nv_d3d10_sharing", "cl_nv_d3d11_sharing
            ","cl_nv_d3d9_sharing", "cl_nv_device_attribute_query", "
            cl_nv_pragma_unroll" ],

        ...
}
```

### A.1.11 PCIe

This section shows how to describe the Peripheral Component Interconnect Express (PCIe) component according to the HPP-DL specification.

**Attributes** Table A.19 shows the specific attributes for PCIe component.

**Transfer rates** Table A.20 shows the PCIe transfer rate depending on the number of lanes.

There are defined four version of the PCIe architecture. Table A.21 shows the PCIe bandwidth per PCIe architecture version used. The raw bit rate is measured on transfers per second.

Table A.19: PCI attributes.

| Name | Type | Description |
|---|---|---|
| class | string | Component type. The value is `pcie`. |
| id | string | Component ID. A PCIe always belong to the hardware platform, so the ID of PCIe will be `platform:0.pcie:<unsigned int>`. |
| description | string | Component description. |
| model | string | Specific model. In this case, it takes the default value `"PCIe"` |
| vendor | string | Vendor of product. In this case, it takes the default value `""` |
| max_speed | magnitude | Maximum transfers per second (GT/s). It is a theoretical measure based on the hardware PCIe specifications (See Table A.20 and Table A.21). |
| max_width | magnitude | Maximum number of lines. |
| average_speed | magnitude | Average transfers per second (GT/s). It is a physical measure obtained by OS. |
| average_width | magnitude | Number of lanes. It could be equal or less than the `max_width` attribute, depending on the CPU internal support. |
| clock | magnitude | Clock of the device. It should be expressed as GHz or MHz. |
| version | string | PCIe version. |
| capabilities | string[] | List of capabilities. |

Table A.20: PCIe transfer rate.

| Number of lanes | Bandwidth per Lane Direction |
|---|---|
| PCIe x1 | 250 Mps |
| PCIe x2 | 500 Mps |
| PCIe x4 | 1000 Mps |
| PCIe x8 | 2000 Mps |
| PCIe x12 | 3000 Mps |
| PCIe x16 | 4000 Mps |

Listing A.10: GPGPU example 2

```
{
        ...

        /* Memory attributes */
        "global_mem_cache_size": 229376,
        "global_mem_cache_type": 2,
        "global_mem_cacheline_size": 128,
        "global_mem_size": 6442123264,
        "local_mem_size": 49152,
        "local_mem_type": 1,
        "host_unified_memory": 0,
        "max_mem_alloc_size": 1610530816,
        "mem_base_addr_align": 4096,
        "svm_capabilities" : 0,
        "max_constant_buffer_size": 65536,

        /* Image related attributes */
        "image_support": 1,
        "image2d_max_height": 32768,
        "image2d_max_width": 32768,
        "image3d_max_depth": 4096,
        "image3d_max_height": 4096,
        "image3d_max_width": 4096,
        "max_write_image_args": 16,
        "max_read_image_args": 256,

        /* Floating-point precision attributes */
        "single_fp_config": 63,
        "half_fp_config": 63,
        "double_fp_config": 63,

        ...
}
```

Table A.21: PCIe architecture.

| PCI Express Type | Raw Bit Rate | Total Bandwidth for x16 Link |
|---|---|---|
| PCIe 1.x | 2.5 GT/s | 40 GT/s (~4 GB/s) |
| PCIe 2.x | 5.0 GT/s | 80 GT/s (~8 GB/s) |
| PCIe 3.0 | 8.0 GT/s | 128 GT/s (~16 GB/s) |
| PCIe 4.0 | 16 GT/s | 256 GT/s (~32 GB/s) |

**Examples** Listing A.12 shows a sample of PCIe specification in HPP-DL.

### A.1.12 FPGA Board

This section shows how to describe the FPGA board component according to the HPP-DL specification.

Listing A.11: GPGPU example 3

```
{
        ...

        /* Vector attributes */
        "native_vector_width_char": 1,
        "native_vector_width_double": 1,
        "native_vector_width_float": 1,
        "native_vector_width_half": 0,
        "native_vector_width_int": 1,
        "native_vector_width_long": 1,
        "preferred_vector_width_char": 1,
        "preferred_vector_width_double": 1,
        "preferred_vector_width_float": 1,
        "preferred_vector_width_half": 0,
        "preferred_vector_width_int": 1,
        "preferred_vector_width_long": 1,

        /* Queue attributes */
        "max_on_device_queues": 1,
        "queue_on_host_properties": 3,
        "queue_on_device_max_size": 262144,
        "queue_on_device_preferred_size": 16384,

        /* Pipe attributes */
        "max_pipe_args": 0,
        "pipe_max_active_reservations": 0,
        "pipe_max_packet_size": 0,

        /* Version attributes */
        "driver_version":"311.50",
        "device_version":"1.1"
}
```

Listing A.12: PCIe example

```
{
        "class":"pcie",
        "id":"platform:0.pci:0",
        "description":"PCI v2 x16",
        "vendor":"",
        "model":"",
        "max_speed":"2.5GT/s",
        "max_width":"x16",
        "average_speed":"2.5GT/s",
        "average_width":"x16",
        "clock":"66 Mhz",
        "version":2,
        "capabilities": []
}
```

**Attributes** Due to the unique programmability of FPGA devices, the attributes of the `FPGA` class represents features of the device driver on the modeled platform rather than features of the FPGA device itself. FPGAs can be programmed to perform almost any task, but their efficiency and interaction with the host is limited by the features offered by the device driver.

Listing A.13: FPGA board example

```
{
        "class": "fpga",
        "id": "platform:0.fpga:0",
        "description": "Xilinx VC707 Evaluation Kit",
        "vendor": "Xilinx Inc.",
        "model": "VC707",
        "rndacc_winsize": 32,/* size of address space in bits, i.e. number of bits
            in addresses */
        "rndacc_latency": 115,/* access latency (ns) */
        "stracc_count": 4, /* number of streams */
        "stracc_throughput": 15.5, /* throughput in MB/s */
        "hwsw_int_latency" : 60, /* average interrupt latency (microseg.) */
        "capabilities" : [ "mastermode", "rndacc", "stracc", "virtaddr", "hwsw_int"
            , "auto_coherency", "scatter_gather" ]
}
```

`General` Table A.45 on p. 170 summarizes the available capabilities of the device. The capability `mastermode` should be set, if the device driver allows memory transfers between host and FPGA without the CPU, i.e. the device can access host memory as bus master. If the device can furthermore use virtual addresses in the host application's virtual address space directly to access host memory, the capability `virtaddr` should be set. If the driver can guarantee that accesses to host memory will be cache coherent the capability `auto_coherency` should be set. If the device can trigger interrupts at the host, the capability `hwsw_int` should be set, and the attribute `hwsw_int_latency` should be set to the average delay between interrupt signalling of the device and acknowledge of the host.

`Random Access` If the device can perform random access on host memory, the capability `rndacc` should be set and the attribute `rndacc_winsize` should be set to the size of the memory window (in bits), in which the device can perform random access on the host memory. In case this size is less than the addressable memory of the system, the capability `rndacc_winmove` should reflect whether or not the driver offers the capability of moving the window in the address space (i.e. change its base address). Furthermore, the attribute `rndacc_latency` should be set to the average delay of non-cached random accesses to host memory.

`Streaming Access` If the device driver offers a streaming interface, the capability `stracc` should be set, the attribute `stracc_count` should be set to the number of parallel streams supported and the attribute `stracc_throughput` should be set to the average data rate of the streaming channel. Furthermore, if scatter/gather access is also supported, the capability `scatter_gather` should be set as well.

Table A.22 summarizes the specific attributes and table A.45 (p. 170) the capability flags for FPGA devices.

**Examples** Listing A.13 shows a sample of FPGA board specification in HPP-DL.

Table A.22: FPGA attributes.

| Name | Type | Description |
|---|---|---|
| class | string | Component type. The value for this kind of object is `fpga`. |
| id | string | Component ID. In this case, `platform:0.fpga:<unsigned int>` |
| description | string | Component description. |
| model | string | Specific model. |
| vendor | string | Vendor of product. |
| hwsw_int_latency | magnitude | average interrupt latency ($\mu$s) |
| rndacc_latency | magnitude | access latency (ns) |
| rndacc_winsize | number | Size of address space in bits |
| stracc_count | number | number of streams |
| stracc_throughput | magnitude | throughput in MB/s |

### A.1.13 DSP Board

This section shows how to describe the DSP board component according to the HPP-DL specification.

**Attributes** Table A.23 shows the attributes for DSP board. These are similar

**Examples** Listing A.14 and A.15 show an example of DSP board example (DSPC-8681).

## A.2 Links

This section shows the rules to describe a link according to the HPP-DL specification.

**Attributes** Table A.24 shows the specific attributes for HPP-DL link entity.

`throughput` attribute. This attribute represents the throughput of a connection using different transfer sizes. It is composed by two values:

- **size**: transfer size used in the connection and it is represented as a magnitude. The default unit is bytes, however it is possible to use another IEC binary prefixes.
- **value**: throughput of the connection expressed in transfer unit size per second (e.g. KiB/s) and it is represented as a magnitude. By default, the value is represented by bytes per second (bytes/s).

Table A.23: DSP attributes.

| Name | Type | Description |
|---|---|---|
| class | string | Component type. The value for this kind of object is `dsp`. |
| id | string | Component ID. In this case, `platform:0.dsp:<unsigned int>` |
| description | string | Component description. |
| model | string | Specific model. |
| vendor | string | Vendor of product. |
| processors | number | Number of processors included on the board. |
| cores | number | Number of cores available between all the processors. |
| pu_num | number | Number of processing units of the board. |
| global_mem_size | magnitude | Represents the size of the memory that exists in the DSP board (`platform:0.dsp:0.memory:0`). Memory size capacity is described using IEC binary prefixes. |
| capabilities | string[] | List of capabilities. Empty for this component |

Table A.24: Link attributes.

| Name | Type | Description |
|---|---|---|
| class | string | HPP-DL object type. The value is `link`. |
| id | string | Link ID. In this case, `link:<unsigned int>`. |
| description | string | Human-readable link description. It could take default value. |
| src_component | string | Source component ID in link. |
| dst_component | string | Destination component ID in link. |
| throughput | throughput[] | Tuple of values that shows the transfer size and its throughput associate. Both are magnitudes |
| latency | magnitude | Access latency. The default unit of measurement is microseconds ($\mu s$). |

Listing A.14: DSP board scheme example

```
{
    /* Definition of DSP board */
    "class":"dsp",
    "id":"platform:0.dsp:0",
    "description":"DSPC-8681",
    "vendor":"ADVANTECH",
    "model":"DSPC-8681"
    "processors":4,
    "cores":32,
    "pu_num":32,
    "capabilities": [...]
},
{
    /* Definition of DSP processor */
    "class": "processor",
    "id": "platform:0.dsp:0.processor:0",
    "description": "Multicore Fixed and Floating-Point Digital Signal Processor",
    "model": "TMS320C6678",
    "vendor": "Texas Instruments",
    "cores": 8,
    "pu_num":8,
    "numa_group":0,
    "capabilities": []
},
/* There are three more processors like this one */
{
    /* Definition of DSP cores */
    "class": "core",
    "id": "platform:0.dsp:0.processor:0.core:0",
    "description": "27.2 GMAC/13.6 GFLOP @ 1.00 GHz",
    "model": "TMS320C66x ",
    "vendor": "Texas Instruments",
    "clock":"1.00 GHz",
    "architecture":"",
    "GFLOPS": 6,
    "GMACS": 27.2,
    "width": 0,
    "pu_num":1,
    "pu_list": [0],
    "capabilities": ["fxp", "flp ",...]
},
/* There are seven more processors like this one for each processor */
{
    "class": "memory",
    "id": "platform:0.dsp:0.processor:0.memory:0",
    "description": "DSP Global Memory",
    "model": "",
    "vendor": "",
    "size": "1 GiB",
    "num_banks":1,
    "capabilities":[]
},
...
```

**Restictions** There are some restrictions in the construction of a link object:

- the source and the destination of a link must be a component,
- there are not links between components of different devices directly. Always,

Listing A.15: Continuing the previous example of DSP board scheme example

```
{
    /* Definition of memory banks */
    "class":"bank",
    "id": "platform:0.dsp:0.processor:0.memory:0.bank:0",
    "description": "DDR3 1333 MHz",
    "model": "",
    "vendor": "",
    "serial": "",
    "slot": "",
    "size": "256 MiB",
    "width": "64 bits",
    "clock": "1333MHz",
    "latency":"",
    "generation":"DDR3",
    "family":"",
    "architecture":"",
    "capabilities":[]
},
/* There are three more processors like this one */
{
    "class":"cache",
    "id": "platform:0.dsp:0.processor:0.core:0.cache:0",
    "description": "L1P",
    "model": "",
    "vendor": "",
    "level": 1,
    "slot": "L1 Cache",
    "size": "32KiB",
    "ways_of_associativity": 0,
    "coherency_line_size":0,
    "number_of_sets":0,
    "physical_line_partition": 0,
    "capabilities": ["data", "write-through"]
},
{
    "class":"cache",
    "id": "platform:0.dsp:0.processor:0.core:0.cache:1",
    "description": "L1D",
    "model": "",
    "vendor": "",
    "level": 1,
    "slot": "L1 Cache",
    "size": "32KiB",
    "ways_of_associativity": 0,
    "coherency_line_size":0,
    "number_of_sets":0,
    "physical_line_partition": 0,
    "capabilities": ["instruction", "write-through"]
},
...
/* The same for each core of each processor */
/* There are three more L2 caches, one for each processor */
```

there is a PCIe component between them of these cases.

Connections covered by HPP-DL links are:

- memory banks and processors,

- core and memory caches of the same processor (e.g. Instruction and data L1 caches),
- caches with other highest level caches,
- processor and caches,
- boards( GPGPU, FPGA, etc. )and PCIe.

**Examples** This section includes different kind of examples: basic and memory architecture examples.

Listing A.16 shows a sample of connection between two different components in HPP-DL specification.

Listing A.16: Link example representing a connection on HPP-DL.

```
{
        "class": "link",
        "id": "link:0",
        "description" : "Link between gpgpu 0 and pcie 0",
        "src_component": "platform:0.gpgpu:0",
        "dst_component":"platform:0.pcie:0",
        "throughput" : [
                {"size":"1", "value":"1 KB/s"},
                {"size":"2", "value":"2 KB/s"},
                /* rest of measures */
        ],
        "latency": "0.8 ms"
}

{
        "class": "link",
        "id": "link:1",
        "description" : "Link between pcie 0 and gpgpu 0",
        "src_component":"platform:0.pcie:0",
        "dst_component": "platform:0.gpgpu:0",
        "throughput" : [
                {"size":"1", "value":"10 KB/s"},
                {"size":"2", "value":"20 KB/s"},
                /* rest of measures */
        ],
        "latency": "0.8 ms"
}
```

Listing A.17 shows an example where throughput and latency are empty, describing only a relationship between source and destination.

Listings A.18 and A.19 represemt the example of SMP architecture shown in Figure 3.4.

Listings A.20 and A.21 represemt the example of NUMA architecture shown in Figure 3.5.

Listing A.17: Link sample between a core and a cache.

```
{
        "class": "link",
        "description" : "Link between core 0 and cache 0",
        "id": "link:2",
        "src_component": "platform:0.processor:0.core:0",
        "dst_component":"platform:0.processor:0.cache:0",
        "throughput" : [],
        "latency": ""
}

{
        "class": "link",
        "description" : "Link between cache 0 and core 0",
        "id": "link:3",
        "src_component":"platform:0.processor:0.cache:0",
        "dst_component": "platform:0.processor:0.core:0",
        "throughput" : [],
        "latency": ""
}
```

Listing A.18: SMP architecture on HPP-DL 1.

```
{

        "class":"hpp"
        ...
        "components":[
                {
                        "class":"platform",
                        "id":"platform:0",
                        ...
                },
                {
                        "class":"memory",
                        "id":"platform:0.memory:0",
                        ...
                },
                {
                        "class":"processor",
                        "id":"platform:0.processor:0",
                        ...
                },
                {
                        "class":"bank",
                        "id":"platform:0.memory:0.bank:0",
                        ...
                },
                ...

        ],
        ...
}
```

Listing A.19: SMP architecture on HPP-DL 2.

```
{
                ...
        "links":[
                {
                        "class": "link",
                        "id": "link:0",
                        "description" : "",
                        "src_component": "platform:0.processor:0",
                        "dst_component":"platform:0.memory:0.bank:0",
                        "throughput" : [
                                {"size":"1", "value":"100 KB/s"},
                                {"size":"2", "value":"200 KB/s"},
                                /* rest of measures */
                        ],
                        "latency": "0.1 ms"
                },
                {
                        "class": "link",
                        "id": "link:1",
                        "description" : "",
                        "src_component": "platform:0.processor:0",
                        "dst_component":"platform:0.memory:0.bank:1",
                        "throughput" : [
                                {"size":"1", "value":"100 KB/s"},
                                {"size":"2", "value":"200 KB/s"},
                                /* rest of measures */
                        ],
                        "latency": "0.1 ms"
                },
                ...

        ]
        ...
}
```

## A.3 Resources

This section shows the rule to describe the resource entity according to the HPP-DL specification.

**Attributes** Table A.25 shows the specific attributes for resource in HPP-DL. It contains a reference to memory components included on HPP-DL specification with the io ports and irq necessary to manage the memory resources on a board (e.g. platform board `platform:0.memory:0` as or FPGA board `platform:0.fpga:0.memory:0`).

**Examples** Listing A.22 shows a sample of two different resources shared/used by other components in HPP-DL. The first example, with id `resource:0`, contains a reference to platform memory (`platform:0.memory:0`), where IRQ and I/O ports information are included. The second example, with id `resource:1`, contains a memory component reference of a FPGA memory (`platform:0.fpga:0.memory:0`).

Listing A.20: NUMA architecture on HPP-DL 1.

```
{

        "class":"hpp"
        ...
        "components":[
                {
                        "class":"platform",
                        "id":"platform:0",
                        ...
                },
                {
                        "class":"memory",
                        "id":"platform:0.memory:0",
                        ...
                },
                {
                        "class":"processor",
                        "id":"platform:0.processor:0",
                        ...
                },
                {
                        "class":"processor",
                        "id":"platform:0.processor:1",
                        ...
                },
                {
                        "class":"bank",
                        "id":"platform:0.memory:0.bank:0",
                        ...
                },
                ...

        ],
                ...
}
```

Figure A.1 shows an example of resource represented on Code A.22.

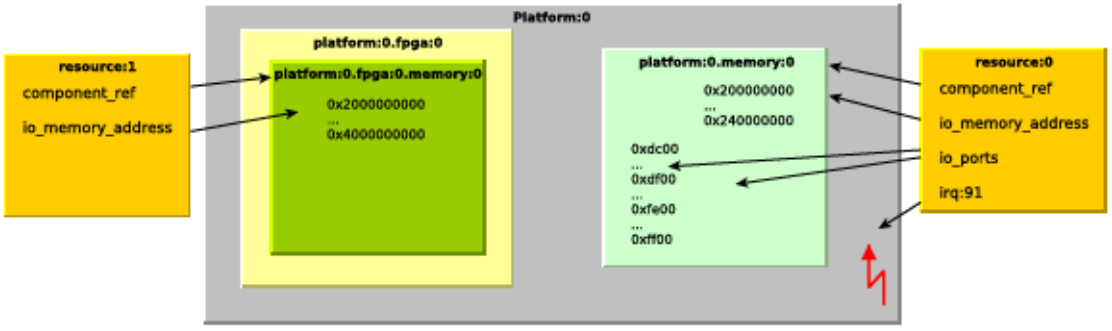


Figure A.1: Sample of resource on HPP-DL

Table A.25: Resource attributes.

| Name | Type | Description |
|---|---|---|
| class | string | HPP-DL object type. The value is `resource`. |
| id | string | Resource ID. In this case, `resource:<unsigned int>`. |
| description | string | Resource description. Its is optional. |
| component_ref | string | It is the reference ID component. It is a reference to a memory component included on a board (FPGA, platform, etc.). |
| io_memory_address | string[] | Memory addresses where data is accessed/located for I/O operations. It is represented as a memory address range, using a dash between the lowest memory address and the highest (e.g. `"0xdc00-0xdf00"`). It could contain one or more values. |
| io_ports | string[] | I/O ports. It is represented as a memory address range, the same way as `io_memory_address`. It could contain zero or more values in case the `component_ref` is not a platform resource. |
| irq | number | IRQ number. It is empty if the `component_ref` is not a platform resource. |

## A.4 Capabilities

This section shows the different capabilities used in the different components.

**Description** A supplementary capability is a feature that has been added to an existing on a component design after the initial introduction of that design to the marketplace. For example, a supplementary capability increases the usefulness of the processor design, allowing it to compete more favourably with competitors and

giving consumers a reason to upgrade, while retaining backwards compatibility with the original design [106].

In HPP-DL, a capability is a string that defines a characteristic of a processing component such as GPGPU, caches or cores. All capabilities are described on this Appendix using a label and a associate description.

An special case of capability is OpenCL feature (see Table A.26). It could be included on cores or gpgpu components.

### A.4.1 OpenCL support

This capability means that openCL is supported by a component. It is included in these components:

- Core.
- Processor.
- FPGA board.
- Integrated GPGPU on a chip or GPGPU board.

OpenCL capability is composed by two values:

- `ocl`, OpenCL string,
- A number which is composed by two numbers split by a dot (e.g. `ocl1.2`).

It can appear zero or several times on capability attributes but never with the same values (e.g. `ocl1.0`, `ocl1.1`, `ocl1.2`, etc.).

Table A.26: OpenCL capabilities.

| Capability | Description |
|---|---|
| `ocl`*N.M* | OpenCL supported. *N* and *M* are the numbers of OCL version (e.g: `ocl1.2`). |

### A.4.2 CPU Cores

Next tables shows these capabilities for Intel, AMD and ARM CPU cores.

Tables A.27, A.28, A.29, A.30, A.31, A.32, A.33, A.34, A.35, A.36, A.37, A.38 and A.39 show the CPU capabilities supported on Intel and AMD cores.

Table A.40, A.40 and A.42 show the capabilities for ARM and ARM64 cores.

Table A.27: Intel-defined CPU capabilities 1.

| Capability | Description |
|---|---|
| fpu | Onboard FPU |
| vme | Virtual Mode Extensions |
| de | Debugging Extensions |
| pse | Page Size Extensions |
| tsc | Time Stamp Counter |
| msr | Model-Specific Registers |
| pae | Physical Address Extensions |
| mce | Machine Check Exception |
| cx8 | CMPXCHG8 instruction |
| apic | Onboard APIC |
| sep | SYSENTER/SYSEXIT |
| mtrr | Memory Type Range Registers |
| pge | Page Global Enable |
| mca | Machine Check Architecture |

Table A.28: Intel-defined CPU capabilities 2.

| Capability | Description |
|---|---|
| cmov | CMOV instructions |
| pat | Page Attribute Table |
| pse36 | 36-bit PSEs |
| pn | Processor serial number |
| clflsh | "clflush" CLFLUSH instruction |
| ds | "dts" Debug Store |
| acpi | ACPI via MSR |
| mmx | Multimedia Extensions |
| fxsr | XSAVE/FXRSTOR, CR4.OSFXSR |
| sse | Streaming SIMD Extensions |
| sse2 | SSE v2 |
| ss | CPU self snoop |
| ht | hyper-Threading |
| acc | "tm" Automatic clock control |
| ia64 | IA-64 processor |
| pbe | Pending Break Enable |

Table A.29: AMD-defined CPU capabilities.

| Capability | Description |
|---|---|
| syscall | SYSCALL/SYSRET |
| mp | MP Capable |
| nx | Execute Disable |
| mmxext | AMD MMX extensions |
| fxsr_opt | FXSAVE/FXRSTOR optimizations |
| pdpe1gb | GB pages |
| rdtscp | RDTSCP |
| lm | Long Mode |
| 3dnow | 3DNow! (AMD vector instructions) |
| 3dnowext | AMD 3DNow! extensions |

Table A.30: Transmeta-defined CPU capabilities.

| Capability | Description |
|---|---|
| recovery | CPU in recovery mode |
| longrun | Longrun power control |
| lrti | Longrun table interface |

### A.4.3 DSP capabilities

Tables A.43 and A.44 include specific capabilities supported by a DSP core [112].

### A.4.4 GPGPU capabilities

The capabilities of a GPGPU are included on `CL_DEVICE_EXTENSIONS` OpenCL attribute [59]. Always, OpenCL capability must be supported.

### A.4.5 FPGA

Table A.45 shows the specific attributes for FPGA capabilities.

### A.4.6 Cache

Table A.46 shows the capabilities for cache component. It specify some restrictions between cache capabilities.

Table A.31: More AMD CPU extension capabilities 1.

| Capability | Description |
|---|---|
| k7 | Athlon |
| k8 | Opteron, Athlon64 |
| mmxext | MMX extensions |
| lahf_lm | LAHF/SAHF in long mode |
| cmp_legacy | If yes HyperThreading not valid |
| svm | Secure virtual machine |
| extapic | Extended APIC space |
| cr8_legacy | CR8 in 32-bit mode |
| abm | Advanced bit manipulation |
| sse4a | SSE-4A |
| misalignsse | Misaligned SSE mode |
| 3dnowprefetch | 3DNow prefetch instructions |
| osvw | OS Visible Workaround |
| ibs | Instruction Based Sampling |

Table A.32: More AMD CPU extension capabilities 2.

| Capability | Description |
|---|---|
| xop | extended AVX instructions |
| skinit | SKINIT/STGI instructions |
| wdt | Watchdog timer |
| lwp | Light Weight Profiling |
| fma4 | 4 operands MAC instructions |
| tce | translation cache extension |
| nodeid_msr | NodeId MSR |
| tbm | trailing bit manipulations |
| topoext | topology extensions CPUID leafs |
| perfctr_core | core performance counter extensions |
| perfctr_nb | NB performance counter extensions |
| perfctr_l2 | L2 performance counter extensions |
| constant_tsc | TSC ticks at a constant rate |
| sse5 | SSE-5 |
| amd_dcm | multi-node processor |

Table A.33: More Intel CPU extension capabilities 1.

| Capability | Description |
|---|---|
| pni | SSE-3 |
| pclmulqdq | PCLMULQDQ instruction |
| dtes64 | 64-bit Debug Store |
| monitor | Monitor/Mwait support |
| ds_cpl | CPL Qual. Debug Store |
| vmx | Hardware virtualization |
| smx | Safer mode: TXT (TPM support) |
| est | Enhanced SpeedStep |
| tm2 | Thermal Monitor 2 |
| ssse3 | Supplemental SSE-3 |
| cid | Context ID |
| fma | Fused multiply-add |
| cx16 | CMPXCHG16B |
| xtpr | Send Task Priority Messages |
| pdcm | Performance Capabilities |
| pcid | Process Context Identifiers |
| dca | Direct Cache Access |

Table A.34: More Intel CPU extension capabilities 2.

| Capability | Description |
|---|---|
| sse4_1 | SSE-4.1 |
| sse4_2 | SSE-4.2 |
| x2apic | x2APIC |
| movbe | MOVBE instruction |
| popcnt | POPCNT instruction |
| tsc_deadline_timer | Tsc deadline timer |
| aes | AES instructions: AES-NI |
| xsave | XSAVE/XRSTOR/XSETBV/XGETBV |
| avx | Advanced Vector Extensions |
| f16c | 16-bit fp conversions (CVT16) |
| rdrand | The RDRAND instruction |
| hypervisor | Running on a hypervisor |
| pebs | Precise-Event Based Sampling |
| eagerfpu | "agerfpu" Non lazy FPU restore |

Listing A.21: NUMA architecture on HPP-DL 2.

```
{
                ...
        "links":[
                {
                        "class": "link",
                        "id": "link:0",
                        "description" : "",
                        "src_component": "platform:0.processor:0",
                        "dst_component":"platform:0.memory:0.bank:0",
                        "throughput" : [
                                {"size":"1", "value":"100 KB/s"},
                                {"size":"2", "value":"200 KB/s"},
                                /* rest of measures */
                        ],
                        "latency": "0.1 ms"
                },
                {
                        "class": "link",
                        "id": "link:1",
                        "description" : "",
                        "src_component": "platform:0.processor:0",
                        "dst_component":"platform:0.memory:0.bank:1",
                        "throughput" : [
                                {"size":"1", "value":"100 KB/s"},
                                {"size":"2", "value":"200 KB/s"},
                                /* rest of measures */
                        ],
                        "latency": "0.1 ms"
                },
                {
                        "class": "link",
                        "id": "link:2",
                        "description" : "",
                        "src_component": "platform:0.processor:0",
                        "dst_component":"platform:0.memory:0.bank:2",
                        "throughput" : [
                                {"size":"1", "value":"1 KB/s"},
                                {"size":"2", "value":"2 KB/s"},
                                /* rest of measures */
                        ],
                        "latency": "0.3 ms"
                },
                {
                        "class": "link",
                        "id": "link:3",
                        "description" : "",
                        "src_component": "platform:0.processor:0",
                        "dst_component":"platform:0.memory:0.bank:3",
                        "throughput" : [
                                {"size":"1", "value":"1 KB/s"},
                                {"size":"2", "value":"2 KB/s"},
                                /* rest of measures */
                        ],
                        "latency": "0.3 ms"
                },
                ...
        ]
        ...
}
```

Listing A.22: Resource examples on HPP-DL

```
{
        "class": "resource",
        "id": "resource:0",
        "description" : "",
        "component_ref": "platform:0.memory:0",
        "io_memory_address": [ "0x200000000-0x24000000" ],
        "io_ports": [ "0xdc00-0xdf00", "0xfe00-0xff00"],
        "irq": 91
}

{
        "class": "resource",
        "id": "resource:1",
        "description" : "",
        "component_ref": "platform:0.fpga:0.memory:0",
        "io_memory_address": [ "0x200000000-0x400000000"],
        "io_ports": [],
        "irq": 0
}
```

Table A.35: Other defined CPU extension capabilities.

| Capability | Description |
|---|---|
| up | smp kernel running on up |
| bts | Branch Trace Store |
| rep_good | rep microcode works well |
| nopl | The NOPL (0F 1F) instructions |
| xtopology | cpu topology enum extensions |
| nonstop_tsc | TSC does not stop in C states |

Table A.36: Auxiliary CPU capabilities.

| Capability | Description |
|---|---|
| ida | Intel Dynamic Acceleration |
| arat | Always Running APIC Timer |
| cpb | AMD Core Performance Boost |
| epb | IA32_ENERGY_PERF_BIAS support |
| xsaveopt | Optimized Xsave |
| pln | Intel Power Limit Notification |
| pts | Intel Package Thermal Status |
| dts | Digital Thermal Sensor |
| hw_pstate | AMD HW-PState |

Table A.37: Intel CPU virtualization capabilities.

| Capability | Description |
|---|---|
| tpr_shadow | Intel TPR Shadow |
| vnmi | Intel Virtual NMI |
| flexpriority | Intel FlexPriority |
| ept | Intel Extended Page Table |
| vpid | Intel Virtual Processor ID |

Table A.38: AMD CPU virtualization capabilities.

| Capability | Description |
|---|---|
| npt | Nested Page Table support |
| lbrv | LBR Virtualization support |
| svm_lock | SVM locking MSR |
| nrip_save | SVM next_rip save |
| tsc_scale | TSC scaling support |
| vmcb_clean | VMCB clean bits support |
| flushbyasid | flush-by-ASID support |
| decodeassists | Decode Assists support |
| pausefilter | filtered pause intercept |
| pfthreshold | pause filter threshold |

Table A.39: New CPU capabilities.

| Capability | Description |
|---|---|
| fsgsbase | RD/WRFS/GSBASE instructions |
| bmi1 | 1st group bit manipulation extensions |
| hle | Hardware Lock Elision |
| avx2 | AVX2 instructions |
| smep | Supervisor Mode Execution Protection |
| bmi2 | 2nd group bit manipulation extensions |
| erms | Enhanced REP MOVSB/STOSB |
| invpcid | Invalidate Processor Context ID |
| rtm | Restricted Transactional Memory |

Table A.40: ARM CPU capabilities 1.

| Capability | Description |
|---|---|
| swp | SWP instruction (atomic read-modify-write) |
| half | Half word support |
| thumb | Thumb (16-bit instruction set) |
| 26bit | 26 Bit Model |
| fastmult | $32x32 \rightarrow 64 - bit$ multiplication |
| fpa | Floating point accelerator |
| vfp | VFP (early SIMD vector floating point instructions) |
| edsp | DSP extensions |
| java | Jazelle (Java bytecode accelerator) |
| iwmmxt | SIMD instructions |

Table A.41: ARM CPU capabilities 2.

| Capability | Description |
|---|---|
| crunch | MaverickCrunch coprocessor |
| thumbee | ThumbEE |
| neon | NEON (second-generation SIMD) |
| vfpv3 | VFP version 3 |
| tls | TLS register |
| vfpv3d16 | VFP version 3 limited to 16 registers |
| vfpv4 | VFP version 4 |
| idiva | SDIV and UDIV hardware division in ARM mode |
| idivt | SDIV and UDIV hardware division in Thumb mode |
| vfpd32 | VFP version for 32 registers |
| lpae | Large Physical Address Extension |

Table A.42: ARM64 CPU capabilities.

| Capability | Description |
|---|---|
| fp | Floating-point |
| asimd | Advanced SIMD |
| evtstrm | Event stream on userspace |

Table A.43: DSP cores specific capabilities 1.

| Capability | Description |
|---|---|
| fxp | Fixed-point arithmetic support. |
| flp | Floating-point arithmetic support. |
| MAPLE-B | Multi-Accelerator Platform Engine for Baseband. |
| Turbo | Turbo decoding. |
| Viterbi | Viterbi decoding. |
| FFT | Fast Fourier transform support. |
| iFFT | Inverse Fast Fourier transform support. |
| DFT | Discrete Fourier transform support. |

Table A.44: DSP cores specific capabilities 2.

| Capability | Description |
|---|---|
| iDFT | Inverse Discrete Fourier transform support. |
| EFCOP | Enhanced Filter Coprocessor. |
| DTF | Data Trace Formatter. |
| ETB | Embedded Trace Buffer. |
| EDMA3 | Enhanced Direct Memory Access 3. |
| SWI | Software interrupt capability. |
| MAC | Multiply-accumulate operation support. |
| FMA | Fused multiply-add, performed with a single rounding. |
| IIR | Infinite impulse response filter |
| FIR | Finite impulse response filter |
| DPIM | Double-Precision Integer Multiplication |

Table A.45: FPGA capabilities.

| Capability Name | Description |
|---|---|
| auto_coherency | driver guarantees cache coherent random access |
| hwsw_int | device can trigger interrupts at host |
| mastermode | device can perform mastermode memory access |
| rndacc | device can access host memory in random access mode |
| rndacc_winmove | device move the base of the random access window to host memory |
| scatter_gather | driver can perform scatter/gather access |
| stracc | device provides streaming interface |
| virtaddr | device can use user-space virtual addresses to access host memory |

Table A.46: Cache capabilities.

| Capability | Description |
|---|---|
| internal | Internal cache. |
| write-through | Writing cache policy where data is written on caches and/or memories at the same time. It is incompatible with `write-back`. |
| write-back | Writing cache policy where data is written on caches and/or memories delayed on time. It is incompatible with `write-through`. |
| instruction | Instruction cache. It is incompatible with `data` or `unified`. |
| data | Data cache. It is incompatible with `instruction` or `unified`. |
| unified | Unify data and instruction caches. It is incompatible with `data` or `instruction`. |